WASHINGTON UNIVERSITY

SEVER INSTITUTE OF TECHNOLOGY



# A SYNTAX-ORIENTED ARTIFICIAL LANGUAGE TRANSLATOR

BY

Laetitia Harrer Snow

Prepared under the direction of Professor W. E. Ball

A thesis presented to the Sever Institute of Washington University in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

June, 1967

Saint Louis, Missouri

WASHINGTON UNIVERSITY
SEVER INSTITUTE OF TECHNOLOGY

---

ABSTRACT

---

# A SYNTAX-ORIENTED ARTIFICIAL LANGUAGE TRANSLATOR

by Laetitia Harrer Snow

---

ADVISOR: Professor William E. Ball

---

June, 1967

Saint Louis, Missouri

---

This paper will present a general description of syntax-oriented artificial language translation. Special reference will be made to the techniques developed by Ingerman, and an implementation of these techniques in PL/I.

Section 2 of this paper presents some of the methods used in describing syntax, semantics, and pragmatics, together with problems encountered both in the language specification, and in the techniques to make use of the specification. The syntax-controlled approach (in which the syntax is not explicitly stated) and some of the advantages and disadvantages of the syntax-oriented techniques are discussed.

Section 3 presents the techniques for syntax-oriented translation developed by Ingerman. The metasyntactic language, and the rationale behind it are discussed, along with a description of the notion of an acceptability table for deciding the applicability of a rule. The metasemantic language and pragmatics are presented. A brief discussion of the parsing and unparsing processors, and of the PL/I program which has been written to implement the process are included.

764

## TABLE OF CONTENTS

TABLE OF CONTENTS

(continued)

# LIST OF TABLES

# LIST OF FIGURES

# A SYNTAX-ORIENTED ARTIFICIAL LANGUAGE TRANSLATOR

## 1. INTRODUCTION

This paper will present a general description of syntax-oriented artificial language translation. Specific reference will be made to the techniques developed by Ingerman (1)*, and an implementation of these techniques in PL/I.

A translator is a processor which accepts input written in a given source language and produces output written in a different target language. In order for such a processor to function, it must have available to it descriptions of the structure of the source language, and of the relationship between the source language and the target language.

A description of the structure of a language is generally known as the syntax of a language. The syntax is used to distinguish whether or not a given subset of the source language input string is "grammatically" correct. If so, a representation of the structure of the input string can then be constructed.

* The numbers in parentheses indicate references in the Bibliography.

A description of the relationship between the source language and the target language must include two parts: (a) the set of possible meanings (in terms of the target language) attributable to each of the syntactic elements; and (b) a statement of exactly which meaning is to be used for each element at a given time during the translation process. The first of these will be referred to as the semantics of the source language in terms of the target language. The second will be referred to as the pragmatics of the source language in terms of the target language.

A syntax-oriented translator is one in which the translation of the source language to the target language is accomplished by using an explicit statement of both the syntax of the source language and the semantics and pragmatics of the source language in terms of the target language. This is in contrast to other methods of translation which have the structure of the source language, and its meaning in terms of the target language, implicitly "buried" in the coding of the translator.

Section 2 of this paper presents some of the methods used in describing syntax, semantics, and pragmatics, together with problems encountered both in the language specification, and in the techniques to make use of the specification. The syntax-controlled approach (in which the syntax is not explicitly stated) and some of the advantages and disadvantages of the syntax-oriented techniques are discussed.

Section 3 presents the techniques for syntax-oriented translation developed by Ingerman (1). The metasyntactic language, and the rationale behind it are discussed, along with a description of the notion of an acceptability table for deciding the applicability of a rule. The metasemantic language and pragmatics are presented. A brief discussion of the parsing and unparsing processors, and of the PL/I program which has been written to implement the process are included.

# 2. METHODS OF TRANSLATION

## 2.1 INTRODUCTION

Despite wide variations in approach, all translation methods have two basic components in common. These are: (a) a description of the syntax, semantics, and pragmatics of the source-target language pair; and (b) a processor which translates an input string using this description.

In practice, translating techniques may be divided into two main categories, syntax-oriented, and what has been called syntax-controlled (2). The terms syntax-oriented and syntax-controlled are not really definitive terms; more appropriate might be explicit syntax and implicit syntax respectively. However, because they are more widely known, the former terms will be used here.

The basic difference in these two categories lies in the relationship between components (a) and (b) above. In syntax-oriented methods, the processor is separate from, and is literally directed by, the formal description of the source-target language pair. In syntax-controlled methods, the description is coded and becomes an integral part of the processor, so that the two components are inseparable.

## 2.2 SYNTAX-ORIENTED TRANSLATORS

### 2.2.1 Definitions and notation

A metalanguage is a language used to describe another language. In particular, a metalanguage whose specific purpose is to describe the syntax of a language is known as a metasyntactic language (or possibly a syntactic metalanguage). Metasemantic languages are defined similarly. A number of metalanguages for description of syntax and semantics have been proposed. For some examples, see references (3) through (9). The most well known of these is Backus Normal Form, a metasyntactic language used to define Algol 60 (3).

The most basic elements in a metasyntactic language are the allowable source language characters, called base objects (or terminal types). In this paper they are written as characters delimited by single quotes (as 'A'). Defined types (abbreviated as d-type) are syntactic elements which are defined in terms of other defined types or base objects. Defined types (or names) are represented by one or more lower case letters; in the case of a multiword name, the words are strung together with hyphens. Syntactic types (abbreviated as s-type) are either defined types or terminal types. The head of the language is that syntactic element which designates the "largest" construction possible in the language.

The structure of a language can be represented as a tree structure, known as a syntax tree. The head of the language is at the top of the tree, and the base objects are at the bottom. Between the head of the language and the base objects are all the defined types of the language. Parsing is the process of breaking up a source language input string into all its constituent defined types according to the syntax of the language. Parsing is analogous to tracing out a path through the syntax tree.

### 2.2.2 A Prototype Metasyntactic Language

A metasyntactic description of a source language is composed of a set of metasyntactic statements. These statements are used to define all the syntactic elements of the language. A skeletal form for such statements which illustrates the essential constituents is

defined type = a sequence of syntactic types [1]

The equal sign serves to separate the defined type on the left from its definition on the right.

In order to represent adequately the relationships which exist between various syntactic elements in a definition, it is necessary to introduce the two operators '+' (for concatenation) and '/' (for alternation). Concatenation denotes the positional relationship between two syntactic elements, as in 'A' + 'B'. This is read as "the character A followed by the

character B". Alternation denotes alternative definitions for the same syntactic type, as in 'A'/'B', which is read as "the character A or the character B".

It is now possible to develop statement [1] more fully. It can be rewritten as

$$\text{d-type} = \text{s-type}_1 / \text{s-type}_2 / \ldots / \text{s-type}_n$$

or [2]

$$\text{d-type} = \text{s-type}_1 + \text{s-type}_2 + \ldots + \text{s-type}_m$$

or a combination of these. This gives more flexibility in stating syntax; for example, the following could be written:

letter = 'A'/'B'/.../'Z'

four-letter-word = letter + letter + letter + letter.

Parentheses are used to indicate logical grouping of syntactic elements, in order to indicate precedence of analysis. One possible form in which the statements [2] might be combined is

$$\text{d-type} = (\text{s-type}_1 + \text{s-type}_2)/\text{s-type}_3 \qquad [3]$$

This statement indicates that 'd-type' has two alternative definitions, either '$\text{s-type}_1$' followed by '$\text{s-type}_2$', or '$\text{s-type}_3$'. As an example, a definition of identifier can be written as

identifier = letter / (identifier + letter) .

This states that 'identifier' will be either 'letter', or 'identifier' followed by 'letter'.

The above example is an illustration of the important concept of recursive definitions. The purpose in this case is to define 'identifier' as a string composed of one or more elements 'letter'. Unless recursive definitions are allowed in the metalanguage as described so far, there is no way to make this definition without limiting the number of elements allowed. For example, the statement

```
identifier = letter/(letter + letter)/
                    (letter + letter + letter)
```

limits the number of elements 'letter' to three. Thus recursion is a compact means of representing the concept of an arbitrary number of elements.

It should be noted that in a recursive definition the number of elements can not be limited. A method for including the basic purpose of recursion, while making it possible to limit the number of elements, is to define a new operator '$'. This operator means essentially "one or more occurrences of ...". The definition of 'identifier' can now be written as

```
identifier = $ letter  .
```

This method is used in the Meta II language (4) (the specifications for which are listed in Appendix 6.1). An extension is to precede the operator '$' by a number which is an upper bound on the number of elements. For example, to limit 'identifier' to a maximum of six elements 'letter', the definition is

```
identifier = 6 $ letter  .
```

A language is called a simple phrase structure language if it satisfies three basic requirements (9). (Equivalent names that are used are "type-2 language" and "context free grammar" (10).) First, if a defined type appears on the right hand side of a statement such as [3], then it must also appear on the left hand side of such a statement. That is, if a defined type is used, it must be defined.

Second, it must be possible to reduce every defined type ultimately to a set of terminal types. This requirement will be more obvious if the notion of the syntax tree is remembered. There must be a path from each defined type to the terminal types at the bottom of the tree.

Third, there must be a "largest" defined type. That is, there must be at least one defined type which does not appear on the right hand side of any statement of the form of [3] above, except possibly the one which defined it. This defined type is the head of the language.

Table 1 gives the syntax specification of a simple language of assignment statements. The syntax is described using the metasyntactic language developed.

### 2.2.3 Semantic Specification

The metasyntactic language developed in the previous section illustrates the construction of syntax statements.

TABLE 1

Rules For a Simple Assignment Statement Language

The rules given here specify a simple language consisting of assignment statements. The rules allow identifiers consisting only of letters, and only integer constants.

1. assignment-statement = left-part + arithmetic-expression

2. left-part = identifier + equal-sign

3. arithmetic-expression = (arithmetic-expression + unary-arithmetic-expression)/ unary-arithmetic-expression/term

4. unary-arithmetic-expression = adding-operator + term

5. term = (term + multiplying-operator + factor)/factor

6. factor = (factor + exponentiation-sign + primary)/primary

7. primary = (left-parenthesis + arithmetic-expression + right-parenthesis)/identifier/ unsigned-integer

8. identifier = (identifier + letter)/letter

9. unsigned-integer = (unsigned-integer + digit)/digit

10. adding-operator = '+'/'-'

11. multiplying-operator = '*'/'/'

12. letter = 'A'/.../'Z'

13. digit = '0'/.../'9'

14. equal-sign = '='

15. exponentiation-sign = '↑'

16. left-parenthesis = '('

17. right-parenthesis = ')'

These statements are used to recognize occurrences of the various defined types in the source language input string. Associated with each syntactic element there is a set of semantic elements which specify the meaning of the source language element in terms of the target language.

It should be noted that the basic ideas of semantic specification are not nearly as well defined as those of syntactic specification. However, there are several characteristics which must exist in all semantic specifications. All that will be given here is an indication of an approach that could be taken to include semantic specification in the metalanguage already developed.

The main function of a translator is to produce output which is based on the input. In a simple case, the output consists of input characters which are merely copied from the input stream, and certain target language constructions (for example, operation codes). Also, it is usually necessary to copy input characters in a different order from that in which they were recognized. This implies that there must be a mechanism for saving input characters for later use.

A simple metasemantic language can now be defined. It consists of the symbols 'OUTPUT' and 'SAVE', each followed by an operand enclosed in parentheses. The symbol 'OUTPUT' flags the operand following as an object to be put into the

output string. The symbol 'SAVE' flags the operand as something to be saved for later reference. The operand can be one or more of the following: (a) one or more target language base objects enclosed in quotes; or (b) a symbol of the form '*-n', which refers to the input characters recognized as the n-th preceding syntactic element in the same statement. A symbolic metasyntactic statement which defines the syntactic type 'a' in terms of syntactic types 'b' and 'c' can be written as

```
a = (b + c)/b  .
```

If symbolic semantics are included, this statement might appear as

```
a = (b SAVE(...) + c SAVE(...))OUTPUT(...)/b OUTPUT(...)
```

To illustrate the use of the proposed metasemantic language further, the following metasyntactic statements can be constructed:

```
statement = identifier + '=' + expression
expression = identifier + '+' + identifier
identifier = 'A'/.../'Z'
```

An example of a statement written according to these rules is 'A=B+C'. A simple target language might include the symbolic operations 'CLA', 'ADD', and 'STO', each of which has a symbolic variable name as an operand. The statements above can be rewritten with semantic specifications as follows:

```
statement = identifier + '=' + expression OUTPUT('STO' *-3
expression = identifier OUTPUT('CLA' *-1) + '+' +
             identifier OUTPUT ('ADD' *-1)
identifier = 'A' SAVE(*-1)/.../'Z' SAVE(*-1)
```

If this metalinguistic specification is applied to the input string 'A=B+C', the resulting target language output will be

```
CLA  B
ADD  C
STO  A
```

The distinction between semantics and pragmatics must be made at this point. In the above example, the set of possible meanings of 'identifier' consists of the metasemantic constructions SAVE(*-1), which are associated with the syntactic types 'A' through 'Z'. This set is called the semantics of the defined type 'identifier'. The pragmatics (a statement of which meaning will be used at a given time) of 'identifier' are implicit in the construction of the metalinguistic statement. That is, by writing a particular semantic construct beside a particular syntactic element, one of the set is chosen to be the particular meaning associated with that syntactic element.

### 2.2.4 Processor Considerations

The most fundamental of the processor-dependent considerations is the question of how the syntax will be used in parsing. There are two approaches, called "top-down"

and "bottom-up". The top-down approach is analogous to assuming the structure of the input string before processing begins, and then attempting to verify that structure. The process begins with the head of the language and works down through the syntax tree to the base objects at the bottom. New characters from the input string are matched against the base objects as necessary. Meta II (4) is a top-down processor. The bottom up approach is analogous to assuming that the structure of the input string is unknown and is to be determined during the processing. The process begins with the characters in the input string, and the base objects at the bottom of the syntax tree. The processor then works up from the bottom of the tree toward the head of the language at the top. The Ingerman translator (1) uses a bottom-up approach.

Another processor-dependent consideration concerns the order in which the metalinguistic statements will be presented to the processor. In most cases, the processor can be constructed in such a fashion that this order is unimportant. However, it is sometimes necessary for a processor to require a certain ordering to accomplish a specific task. A hypothetical example is to have the statements in alphabetic order by defined type. This notion of ordering does not affect the meaning of the statements. It merely simplifies some processor operation.

A further consideration concerns the order of alternative definitions for the same syntactic element. A processor must have some fixed order in which these alternatives are to be applied. The usual convention is to apply them in a left-to-right fashion. From the point of view of writing syntactic specifications this presents a problem. Consider a symbolic metasyntactic statement such as

a = b/(b+c)

which states that the defined type 'a' can be either the syntactic type 'b' or the construct 'b' followed by 'c'. If the input string is examined and the occurrence of syntactic type 'b' is found, the definition of 'a' has been satisfied. However, the second alternative will never be reached and thus the construct '(b+c)' will never be found. This statement must be rewritten as

a = (b+c)/b .

It should be noted that some metalanguages (e.g., the Ingerman metasyntactic language) do not provide a distinct alternation operator, but rather employ two or more definitions for the same defined type. The alternative definitions are interpreted in some fixed order by the processor, say, from the top down. The same care must be taken in this scheme to avoid the problem of misordered alternatives cited above.

Another processor problem occurs when recursive definitions are used in a top-down processor. This is known as

the "left recursion" problem. As an example, consider the following definition of 'identifier',

identifier = (identifier + letter)/letter .

In this example, a top-down processor, in attempting to recognize the defined type 'identifier', will loop indefinitely trying to recognize 'identifier'. A possible solution is to replace recursion with the special symbol '$' mentioned in section 2.2.2.

Another processor-dependent consideration has to do with the problem of failing to find a certain expected syntactic element. This is known as the "back-up" problem. The processor contains a pointer to the current syntactic element being added to the parsing tree. The term back-up refers to the backing up of this pointer to some previous step in the analysis when a failure occurs. As an example, suppose that an input string is being examined for the occurrence of a construct such as:

$$\text{d-type} = \text{s-type}_1 / \text{ s-type}_2 / \ \ldots \ / \text{ s-type}_n \quad ,$$

where

$$\text{s-type}_i = \text{s-type}_{i,1} + \ \ldots \ + \text{s-type}_{i,m} \quad .$$

At some stage the input string has been successfully matched with the given definitions up through '$\text{s-type}_{i,j-1}$', but upon examining the input string for the occurrence of '$\text{s-type}_{i,j}$', the matching fails ($j \leq m$). Either the input string must

be declared ungrammatical, or the processor must back up to some previous point in the analysis and try an alternative definition. In some cases, the source language is such that back-up is not necessary (e.g., Meta II). This is the case when each syntactic unit being sought has a unique initial character. In most cases, however, the complexity of the language is such that back-up is necessary. The problems associated with back-up include not only the backing up of the processor to try a new goal, but also the backing up of the pointer to the corresponding previous source language character. Additionally, any output which may have been generated while tracing out the path which failed must be eliminated.

The last processor-dependent consideration is the way in which the processor recognizes the basic character set of the source language. The syntax analysis phase of a translator really has two parts, a recognizer and an analyzer. The most basic recognizer is just a character picking routine, whose function is to supply the analyzer with a steady flow of input characters on which to operate.

A slightly more complex recognizer could be given the task of setting a switch to tell the analyzer what kind of character this one is: a letter or a digit or a special character. A much more sophisticated recognizer might be given definitions of 'identifier' and 'number', and might

recode these as simple operands for the analyzer (recoding them in a table and giving the analyzer a pointer to the table, for example). Thus it can be seen that the two parts of the translator can stand in varying relationships to one another. The analyzer may have a basic symbol set which is actually some levels removed from the basic character set of the machine on which it is operational. In this case, the analyzer need be concerned only with kinds of things, not the things themselves.

## 2.3 SYNTAX-CONTROLLED TRANSLATORS

Syntax-controlled translators do not use the syntax of the source language in an explicit form. The original description of syntax, semantics, and pragmatics has been coded into some sort of tabular form, at least. They may, in some cases, be completely interwoven into the actual coding of the translator.

Syntax-controlled translators are tailored to a particular source-target language pair. Thus it is not possible to describe in any detail a general approach that may be taken. What will be described here are two approaches to the problem, using bounded context and operator precedence translation schemes as examples.

Bounded context translation implies that at any given point during the translation process, the next step to be

taken is determined by the current source language element and N elements on either side of the current element, where N is dependent on the structure of the source language (11). The same kind of thing can be said about syntax oriented methods: the action to be taken at any point depends on the current input element, and the structural relationships between all the previous input elements. However, in bounded context translation there is a fixed upper bound on the number of elements necessary to identify the input structure.

A very simple example of bounded context translation can be constructed for translation of an arithmetic expression from (correct) complete parenthesis form to leading operator Polish form. (Complete parenthesis form is a notation in which all parentheses indicating hierarchy of operators are included.) A set of rules for this translation, using a left to right scan of the input, and only binary operators, is:

1. If the input symbol is not a ')', continue the scan.
2. If the input symbol is a ')', recode the preceding operand-operator-operand triplet as an operand of the form operator-operand-operand, and continue the scan.

The usefulness of this scheme is limited, since one prefers to be able to use unary operators, and prefers not to use so many parentheses. However, it can be seen that, for this example, N is zero. That is, the action to be taken depends on only the current input element, and on none on either side of it.

This scheme can be enlarged slightly by allowing the rules to specify "correctness", that is, that the operator-operand relationship must be of the form '(' operand operator operand ')'. Then the rules become

1. If the input symbol is not a ')', continue the scan.
2. If the input symbol is a ')', and the four preceding input symbols are:
   a. '(' operand operator operand, then recode them as an operand of the form operator-operand-operand and continue the scan.
   b. otherwise, stop.

In this case, N = 4, since at most four elements to the left of the current input symbol are necessary to determine the action to be taken.

These informal rules can be rewritten in a more formal manner using Floyd's productions (8), as follows:

| | Production | Action |
|---|---|---|
| 1. | ( S $\Theta$ S )$\Delta$ $\rightarrow$ $P_i$ $\Delta$ | Code $P_i$ as $\Theta$ S S, i = i+1 |
| 2. | S S S S )$\Delta$ $\rightarrow$ empty | Stop |
| 3. | S $\Delta$ S = S S $\Delta$ | |

Here 'S' denotes any symbol whatsoever, $\Theta$ denotes any binary operator, and $P_i$ a list of recoded operand-operator-operand triplets. '$\Delta$' designates the position of the input pointer, where the current input symbol is just to the left of it.

The symbol '→' can be read "produces" or "becomes", applied to the input string. The set of productions is applied to the input string from the top down. When a production applies and the appropriate action has been taken, one returns to the top of the set and repeats the process.

This is a compact and precise notation in which to write the rather wordy rules given before. Production 1 states the criterion of recoding, and the recoding to be done. Production 2 is the "correctness" condition. Production 3 moves the input pointer, that is, it "continues the scan".

Applied to the expression ((B*C)+((D*E)-F)), the above productions would generate the set of $P_i$ as

$$P_1 = *BC \quad P_2 = *DE \quad P_3 = -P_2F \quad P_4 = +P_1P_3 \quad ,$$

which is the leading operator Polish string +*BC-*DEF .

This sort of production scheme, recoded into a more amenable form for computer use, has formed the basis for a number of translators. It operates over a limited subset of the input string, and each action performed is dependent only on that subset. The emphasis here is not on the structure of the source language as a whole, but on the bounded subset with which each separate production is concerned.

Another class of translation scheme often used in syntax-controlled methods is that of precedence analysis (12) (13), which is based on the precedence relations which hold among

the various operators in the language. This is a powerful method, and has been widely used in varying forms in many translators. However, it requires that the structure of the source language be implicit in the coding of the translator, with the possible exception of the precedence relations, which may appear in tabular form.

The basic idea of operator precedence analysis is illustrated by an example of a scheme for translating simple assignment statements to reverse Polish strings. The allowable operators are the source characters ';', '=', '+', '-', '*', '/', and '↑', from lowest priority to highest. The only other allowable characters are single-letter variable names. The source statements are scanned from left to right by a processor which has one push down stack, called 0, used for stacking operators.

The following procedure will generate as output the reverse Polish string representation of the input string. The top of 0 is denoted by $0_t$. The priority of an operator $\alpha$ is designated $P(\alpha)$. The current input character during the scan is denoted by the symbol $S_i$. $0_t$ is initialized to the null operator, which has a priority lower than any other operator.

1. If $S_i$ is an operator

    a. If $P(S_i) \leq P(0_t)$, put $0_t$ in the output, pop up 0.

i. If $0_t$ is the null operator, stop.

ii. Otherwise, go to 1.a to repeat the test.

b. If $P(S_i) > P(0_t)$, push down 0, put $S_i$ in $0_t$, go to 1 to continue scan.

2. Otherwise, put $S_i$ in the output, go to 1 to continue the scan.

If the above scheme is applied to the statement A = B * C + D * E - F;, the reverse operator Polish string ABC*DE*+F-= will be generated.

This type of scheme (in a much more complex form, of course) has been employed in the Algol 60 translator described in (13). In that implementation, all source language delimiters are given priority values. The source language translated is full Algol 60 (3), with only minor exceptions (no dynamic own arrays or integer labels are allowed, and all formal parameters must have specifications).

## 2.4 DISCUSSION OF SYNTAX-ORIENTED METHODS

The most compelling aspect of syntax-oriented methods is that they augment the concept of the computer as a general purpose machine, by providing a base for the study of computer languages. As an exploratory tool, a syntax-oriented translator may be used in the design of new languages, or in the expansion, alteration, or replacement of source or target languages in the study of existing languages. It may be used

as an experimental compiler for the language under study. It provides a possible teaching aid in the explicit and precise statement of the characteristics of a source-target language pair. Finally, by its generality, it allows description of as large or as small a structure as desired. Thus any size "language" might be defined, from expression to statement to procedure to program.

Syntax-oriented methods make bootstrapping fairly easy. Bootstrapping is the process by which a translator running on one machine can be used to produce the same translator to run on another machine. Suppose machine A has a syntax oriented translator written in source language L, and that the translation of L to some machine language can be accomplished using translator T. The translator T requires two things: the metalinguistic specifications for a translation, and the source language to be translated. If the source form of T is translated using metalinguistic specifications for the L-to-machine-language-B translation, the result will be a translator T which will run on machine B.

Syntax-oriented methods can be used as a possible means for investigation of the practicality of machine-independent translation. Machine-independence denotes what its name implies: independence from a particular machine. In using a translator, it is often desirable during the translation to be able to ignore some of the machine-dependent features

of the target language. A particular procedure-oriented language can be translated into some intermediate machine-independent form, which is designed to be easily translated into machine language for any particular machine. This has been a proposed method of operation for many years (14). The solution to the m language, n computer problem (which seems somewhat remote) could come from a scheme such as this. All procedure-oriented languages would be translated into some universal computer-oriented language, which could in turn be assembled or interpreted on the existing computers.

It should be noted that syntax-oriented methods are actually at a higher level than syntax-controlled methods. A syntax-controlled translator is constructed for one source language and one target language. A syntax-oriented translator is constructed for one metalanguage, which will permit description of a number of source-target language pairs. Lest we grow too enthusiastic about syntax oriented methods, however, let us consider some of the more obvious disadvantages and criticisms.

The main criticism of syntax-oriented methods is simply that they are inefficient. This has not been proved conclusively, and may not be provable. However, it seems intuitive that the more general a method, the more inefficiency there will be for one specific use of the method. (Consider, for example, OS/360.) The most frequently cited reason for

the inefficiency is that syntax-oriented methods will waste a great deal of time traversing some path through the syntax tree which ultimately turns out to be the wrong path (the "you can't get there from here" problem). Some progress has been made in this area, with the development of a technique for deciding whether or not a particular rule is applicable before actually applying it (applicable in the sense that use of it can lead to the current goal of the processor) (1). Even so, general inefficiency with respect to any one translation seems to be a reasonable assumption. It should be noted that syntax-controlled methods, since they are generally tailored to the source-target language pair involved, may be made as efficient as the programmer desires (or is able).

Another aspect of efficiency which may be considered is the efficiency of the resulting target language that is produced by a translator. Some of the more common means of achieving some measure of efficiency are (a) elimination of recomputation of common subexpressions; (b) the removal of invariant computations in a loop to outside the range of the loop; (c) the elimination of redundant instructions; and (d) the efficient utilization of hardware such as accumulators and index registers. The point has been made (9) that target code generated during a syntax-oriented analysis of the source language cannot be efficient, because it is difficult to provide for the above optimization of the target code. It

is possible that a partial solution to this problem might be to generate target code that is suitable for optimization by a later processor.

Another complaint about syntax-oriented methods is the difficulty in producing source language diagnostics when the input string is ungrammatical. There have been several partial solutions to this problem cited as having been tried (9). Among them is a scheme (15) which, when presented with an incorrect input string, will make "local" insertions and deletions in the input string until a syntactically correct string is produced.

# 3. INGERMAN'S TRANSLATOR

## 3.1 INTRODUCTION

This section describes the translator which has been written to implement the techniques developed by Ingerman (1). The presentation here is a fairly general one, with specific attention paid to the basic philosophy used in the parsing processor. The notation used is that used in the implementation. For more details of any particular point, consult either the description and listing of the PL/I program in Appendices 6.2 and 6.5, or reference (1).

The translator consists of three parts: a metalanguage input routine, the syntax analyzer (or parsing processor), and the semantic output generator (or unparsing processor). The basic philosophy used in the translator is to analyze the source language input string according to the syntactic rules provided to the translator. Then, if the input string is determined by the processor to be syntactically correct, the translator will generate target language output based on the analysis of the input string, and the rules of semantics and pragmatics which are also provided to it.

The metalanguage input routine has the task of reading in the rules, written in the syntactic and semantic metalanguages. It also constructs a table called the acceptability table, for use by the parsing processor.

The parsing processor has two purposes. The first is to determine the correctness of the source language input string. The method by which the parsing processor analyzes the input string is basically a "bottom-up" method. The processor selects a goal which is one of the defined elements in the syntactic rules (initially the head of the language). It then attempts to follow a chain of syntactic rules from the current base objects in the input string toward the goal. As the chain of rules builds up, more base objects from the input string are required by the rules, and sub-goals at lower levels of the parsing tree are established. For a valid construction, the head of the language is reached at the same time the source language input string is exhausted. The parsing processor makes use of the acceptability table, which is constructed from the syntactic rules. It is used to decide whether or not the parsing processor can reach the current goal from the current position in the syntax tree. This table can save a great deal of "useless" time spent tracing out rule chains which will ultimately fail because they cannot possibly lead to the current goal.

The second purpose of the parsing processor is to provide the unparsing processor with a complete description of the relationships between all the base objects in the input string. It does this by constructing a list of the rules which have been successfully applied, which is then passed on to the unparsing processor.

The unparsing processor has as its function the actual translation according to the rules of semantics and pragmatics that are supplied to it. These are directly associated with their respective syntactic rules used in the parsing processor. Thus the list of rules applied by the parsing processor gives the order of rules to use in unparsing. The semantic and pragmatic rules actually provide the "meaning" of the source language constructs in terms of the target language. The term "meaning", in the framework of a translator, denotes the production of output which represents the mapping of the source language onto the target language, according to the syntactic, semantic, and pragmatic rules supplied.

The lowest level of semantic specification consists of base objects which are simply to be placed as is in the output string, and various special characters which control the spacing of the output. The next level consists of metasemantic constructs which are called clauses. Clauses actually refer, within one metasemantic rule, to the relationship of the various metacomponents of the metasyntactic rule which was used to analyze the string. Combinations of base objects and clauses of varying complexity are used to specify the output, which might be, for example, symbolic machine language.

In summary, the translator consists of an input routine, and two main phases, the parsing processor and the unparsing processor. The parsing processor has two purposes: to

determine the correctness of the input string, and to produce a parsing tree. This tree contains a complete description of the interrelationships between all the objects in the input string. These interreleationships are then used by the unparsing processor to determine the set of meanings which may be attributed to the objects, and to produce as output the particular meaning selected by the pragmatics.

## 3.2 SPECIFICATION OF SYNTAX

The formation of metasyntactic rules is based on the notion of base objects and defined types discussed in Section 2.2. The basic formation of a metasyntactic rule, is, from left to right, one or more metacomponents, followed by the metaresult which they define. Metacomponents may be either base objects, which will be represented here as one character delimited by single quotes, or defined types, which will be represented here as one to four capital letters (to correspond to the implementation). A simple rule to define the character 'A' as a 'letter' might be written as

'A' L

where the name L is an abbreviation understood to mean 'letter'. It is immediately obvious that a base object cannot be a metaresult. A rule may have more than one metacomponent, as in the rule

LPL AE AS

which defines an 'arithmetic-statement' as being a 'left-part-list' followed by an 'arithmetic-expression'.

Base objects which are considered to have no explicit meaning within the syntactic structure, but are simply objects which are expected to appear in the input string at certain points, may be used as metacomponents, as in the rule

ID '=' LP

which states that an 'identifier' followed by the object '=' is a 'left-part'. On the other hand, if a base object is to be referred to in a more general way, then it should be given a name, as in the two rules

'+' AO
'-' AO

which state that the objects '+' and '-' are both to be known as 'adding-operator'.

Recursive rules may be constructed; that is, a defined type which is a metaresult in a certain rule may be a metacomponent of that same rule. For example, a rule which might partially define 'identifier' is

ID L ID .

It should be noted that the rules must satisfy the conditions stated in Section 2.2: (1) there must be at least one metaresult which is not used as a metacomponent in any rule (this is the head of the language); (2) all metaresults must be reducible to a set of base objects (there is a stronger statement of this "grounded" condition which will be made shortly); (3) all metacomponents which are defined types must be used as metaresults in at least one rule.

The set of syntactic rules for a simple assignment statement is given in Table 2. It is a direct translation of the rules given in Table 1. Table 3 contains a listing of the same rules in a proper order for the Ingerman processor. (This is a requirement of the current implementation. See Appendix 6.2 for details.) These rules allow identifiers consisting only of letters, and only integer constants. Otherwise they are the same rules that are incorporated in the larger (but still simple) language listed in Appendix 6.2.

It has been stated that the parsing processor determines a goal (initially the head of the language) and attempts to build a chain of rules from the current source language input string to that goal, establishing sub-goals and examining input characters as necessary. Having described how to form rules of syntax of the source language, it is now necessary to consider the notion of a rule chain, in order to describe the construction of the acceptability table.

A rule chain may be defined as "a sequence of rules such that the metaresult of the k-th rule is one of the metacomponents (or the sole metacomponent) of the (k+1)-st rule" (1). A rule chain is said to have a loop in it if the metaresult of one of the rules is a metacomponent either of a rule preceding it in the rule chain, or of itself. If a rule chain has a loop in it, at least one metacomponent of one of the rules in the loop must require a new base object from

TABLE 2

LANGUAGE OF ASSIGNMENT STATEMENTS
RE-WRITTEN IN INGERMAN'S NOTATION.

```
 1     LP   AE   AS
 2     ID   EQ   LP
 3     UAE  AE
 4     AE   UAE  AE
 5     T    AE
 6     AO   T    UAE
 7     F    T
 8     T    MO   F    T
 9     P    F
10     F    ES   P    F
11     ID   P
12     UI   P
13     LPN  AE   RPN  P
14     L    ID
15     ID   L    ID
16     D    UI
17     UI   D    UI
18     '+'  AO
19     '-'  AO
20     '*'  MO
21     '/'  MO
22     'A'  L
23     'B'  L
24     'C'  L
25     'D'  L
26     'E'  L
27     'F'  L
28     'W'  L
29     'X'  L
30     'Y'  L
31     'Z'  L
32     '0'  D
33     '1'  D
34     '2'  D
35     '3'  D
36     '4'  D
37     '5'  D
38     '6'  D
39     '7'  D
40     '8'  D
41     '9'  D
42     '='  EQ
43     '|'  ES
44     '('  LPN
45     ')'  RPN
```

NOTE: THESE SYNTACTIC RULES ARE IN THE SAME ORDER AS TABLE 1. THIS IS A 'TOP DOWN' ORDERING, AND IS NOT ACCEPTABLE FOR THE INGERMAN PROCESSOR.

ABBREVIATIONS USED:

| | |
|---|---|
| LP | LEFT-PART |
| AE | ARITHMETIC-EXPRESSION |
| AS | ASSIGNMENT-STATEMENT |
| ID | IDENTIFIER |
| EQ | EQUAL-SIGN |
| UAE | UNARY-ARITHMETIC-EXPR. |
| T | TERM |
| AO | ADDING-OPERATOR |
| F | FACTOR |
| MO | MULTIPLYING-OPERATOR |
| P | PRIMARY |
| ES | EXPONENTIATION-SIGN |
| UI | UNSIGNED-INTEGER |
| LPN | LEFT-PARENTHESIS |
| RPN | RIGHT-PARENTHESIS |
| L | LETTER |
| D | DIGIT |

TABLE 3

LANGUAGE OF ASSIGNMENT STATEMENTS IN PROPER ORDER FOR INGERMAN PROCESSOR.

| | | | | |
|---|---|---|---|---|
| 1 | LP | AE | AS | |
| 2 | ID | EQ | LP | |
| 3 | ID | L | ID | |
| 4 | ID | P | | |
| 5 | UAE | AE | | |
| 6 | AE | UAE | AE | |
| 7 | AO | T | UAE | |
| 8 | F | ES | P | F |
| 9 | F | T | | |
| 10 | T | MO | F | T |
| 11 | T | AE | | |
| 12 | P | F | | |
| 13 | UI | D | UI | |
| 14 | UI | P | | |
| 15 | LPN | AE | RPN | P |
| 16 | L | ID | | |
| 17 | D | UI | | |
| 18 | '+' | AO | | |
| 19 | '-' | AO | | |
| 20 | '*' | MO | | |
| 21 | '/' | MO | | |
| 22 | 'A' | L | | |
| 23 | 'B' | L | | |
| 24 | 'C' | L | | |
| 25 | 'D' | L | | |
| 26 | 'E' | L | | |
| 27 | 'F' | L | | |
| 28 | 'W' | L | | |
| 29 | 'X' | L | | |
| 30 | 'Y' | L | | |
| 31 | 'Z' | L | | |
| 32 | '0' | D | | |
| 33 | '1' | D | | |
| 34 | '2' | D | | |
| 35 | '3' | D | | |
| 36 | '4' | D | | |
| 37 | '5' | D | | |
| 38 | '6' | D | | |
| 39 | '7' | D | | |
| 40 | '8' | D | | |
| 41 | '9' | D | | |
| 42 | '=' | EQ | | |
| 43 | '\|' | ES | | |
| 44 | '(' | LPN | | |
| 45 | ')' | RPN | | |

the input string, or the processor will loop indefinitely. (This is the stronger statement of the "grounded" condition referred to previously.) It should be noted that the only rule in a rule chain which may have only base objects as metacomponents is the leftmost rule.

As an example, consider the (sub)set of rules

| | | | |
|---|---|---|---|
| 1. | 'A' | L | |
| 2. | ID | L | ID |
| 3. | L | ID | |
| 4. | ID | '=' | LP |
| 5. | LP | AE | ST |

If the current goal is ST, and the current input character is 'A', then the rule chain 1-3-4-5 is one of the possible rule chains from the current input character to the current goal.

In the same example, another rule chain from 'A' to ST is 1-3-2-4-5. This rule chain contains a (simple) loop, since rule 2 has ID as both a metacomponent and the metaresult. Less trivial loops can be found in the rules of Table 1. For example, the rule chain 3-5-6-7 forms a loop containing 'arithmetic-expression', 'term', 'factor', 'primary', 'arithmetic-expression'. This same chain also contains other loops.

## 3.3 THE RULE INPUT ROUTINE

The rule input routine reads in the metalinguistic statement of the translation desired. Since it is a rather

clumsy process to work with the rules in their character form, they are recoded into a form more suitable for processing. This is done by constructing a list, P, of all the metacomponents used in the rules, both base objects and defined types. The rules themselves are recoded by substituting a pointer to the appropriate element of P for each element of each rule. Corresponding to P are several auxiliary lists which provide information such as (a) whether or not this element is used as a leftmost metacomponent, and if so, which rule is the first rule in which this is so; (b) whether or not this element is a base object or a defined type; (c) whether or not this element is the metaresult of some rule, and if so, what is the corresponding column in the acceptability table.

The rule input routine also constructs the acceptability table, which is designed to answer the following two questions which the parsing processor will ask. (i) Is there a rule chain with this rule as the leftmost rule that leads only through leftmost metacomponents to the current goal? (ii) Is there a rule chain from this rule (which has more than one metacomponent) to the current goal?

The acceptability table is a Boolean matrix. The column headings of the matrix are all the defined types which are used as metaresults. The row headings are all the metacomponents, both base objects and defined types. The matrix is

constructed in the following way. Every rule is examined, and a one put in each matrix element that has as its row heading the leftmost metacomponent of the rule, and as its column heading the metaresult of the rule. Then each column of the matrix is examined. If there is a one in a column, then that row is logically OR-ed with the row which corresponds to that column of the matrix. The matrix thus produced will answer the question, "Is there a rule chain with this rule as the leftmost rule which leads only through leftmost metacomponents to the current goal?"

In order to answer the second question, it is necessary to treat each non-left metacomponent which is a defined type as a separate element. A new row is added to the matrix for each use of each non-left metacomponent, and a one put in the column headed by the metaresult of that rule. The OR-ing procedure is applied to the additional rows of the matrix. This matrix will now answer the second question, "Is there a rule chain from this rule (which has more than one metacomponent) to the current goal?" In addition, each use of a metacomponent as a non-left metacomponent must be added to the list P as a new defined type, and the coding of the particular rule changed to point to this new element of P, rather than the original one. Thus each use of a metacomponent as a non-left metacomponent is treated as if the metacomponent were a new and different defined type in the language.

## 3.4 THE PARSING PROCESSOR

The basic philosophy of the parsing processor is to trace a path from the base objects found in the input string through the syntax tree to the head of the language at the top of the tree. The processor is initialized by choosing the initial goal to be the head of the language. The starting point for the search for a rule chain to the goal is chosen to be the first character of the input string. Before a more detailed description of the process is given, two important points should be discussed.

The first point concerns the rationale followed in choosing a new rule to apply. When it is necessary for the parsing processor to choose a new rule, it will know one of two things: either the current character in the input string, or the last rule successfully used (that is, the last defined type recognized in the input string). Since the metasyntactic rules are constructed in a left-to-right fashion, an obvious rule to apply is one in which the leftmost metacomponent is either the current input character or the last defined type successfully recognized. At this time the acceptability table can be used to determine the possible applicability of this rule. (The acceptability table was constructed to answer the question, "Is there a rule chain (with this rule as the left-most rule) which leads only through leftmost metacomponents

to the current goal?") The acceptability table is entered with the leftmost metacomponent of this rule as the row index, and the current goal as the column index. If the selected element is a one, then a rule chain exists, and this rule is at least a possible link in the rule chain.

As an example, consider the following rule, which defines the metaresult 'MR1' to be a series of two metacomponents:

MC1 MC2 MR1

If 'MC1' is the metaresult of the last rule successfully applied, and 'MR1' is the current goal, then the acceptability table will indicate that this rule is applicable.

The second point concerns the further processing of a rule after it has been determined to be possibly applicable. The acceptability table has already provided the information that it is possible to reach the current goal from a rule which has a particular leftmost metacomponent. If a rule has more than one metacomponent, it is possible that one of the non-left metacomponents is not applicable (and thus that the whole rule is not applicable). Thus these non-left metacomponents must be examined. Metacomponents which are base objects are simply compared to the current input string character to determine if this rule applies. Metacomponents which are not base objects must be examined further. In fact, a non-base object, non-left metacomponent must be established as a new (sub)goal, and the input string examined for the

occurrence of this defined type. However, the acceptability table was constructed to answer the additional question, "Is there a rule chain from this rule (which has more than one metacomponent) to the current goal?" Thus, before the processor actually examines the input string, the acceptability table can be entered with this use of this (non-leftmost) metacomponent as a row index, and the current goal as a column index. If the selected element is a one, then there is a rule chain to the goal which includes this rule (and this non-leftmost metacomponent).

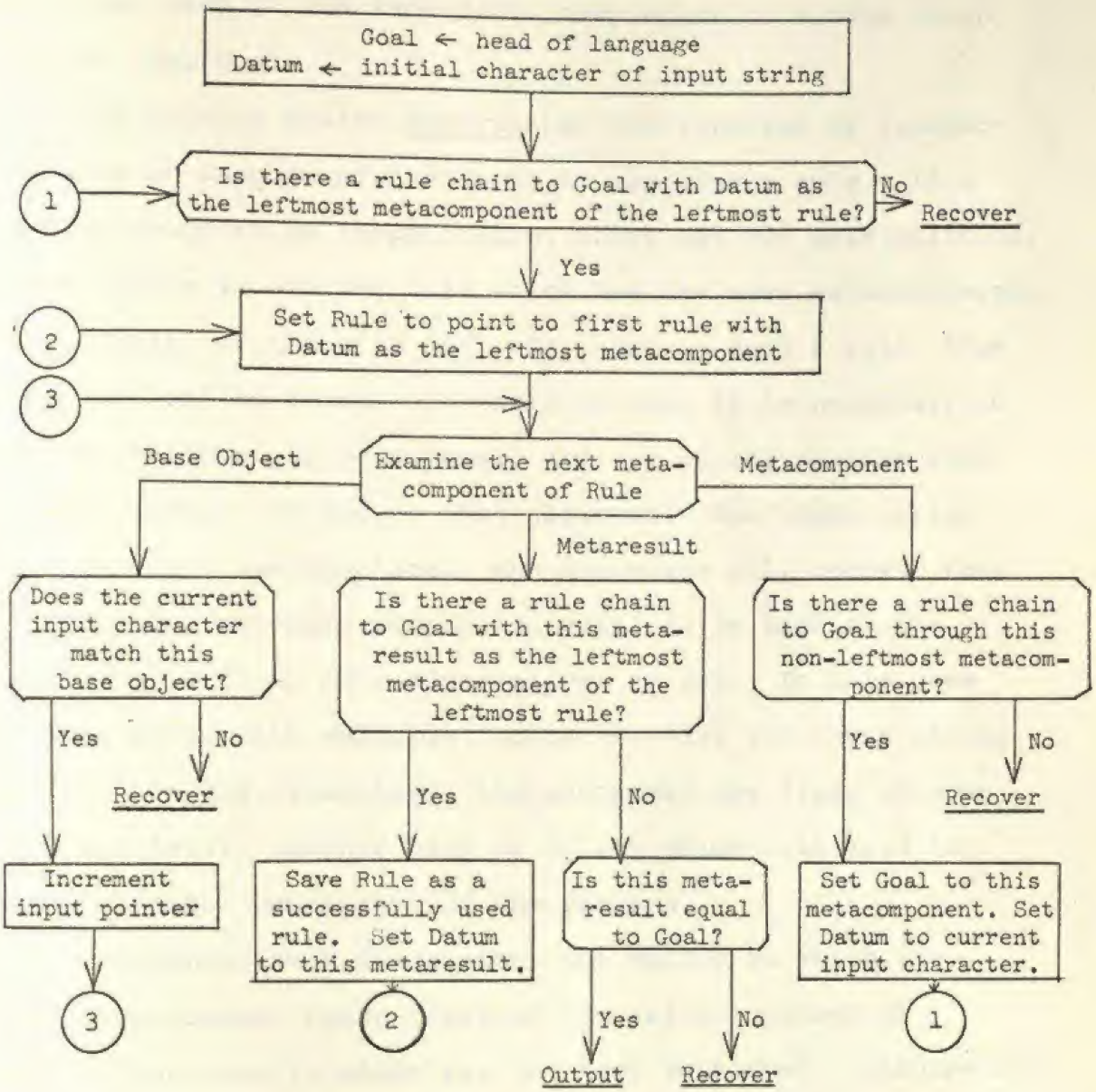
As an example, consider the following two rules defining the metaresults 'MR1' and 'MR2', both of which have the same left-most metacomponent.

Rule 1: MC1 MC2 MR1

Rule 2: MC1 MC3 MR2

If 'MR2' is the current goal, and 'MC1' the last metaresult successfully found, the acceptability table will indicate (in answer to the first question) that Rule 1 is applicable, because there is a rule chain (with MC1 as a leftmost metacomponent of the leftmost rule) to the goal MR2. It is not until the second metacomponent of Rule 1 is examined that it is found that the rule is not applicable.

A flowchart of the philosophy of the parsing processor is given in Figure 1. This flowchart omits most of the details of the processor. It is intended to indicate the logical steps taken by the processor. The flowchart is self-contained, except for two "exits", Output and Recover.




A Flowchart of the Basic Parsing Philosophy

FIGURE 1

The routine called Output has the function of recording the rules used in the successful completion of a rule chain to some (sub)goal.

The routine called Recover has the function of recovering from an unsuccessful attempt at applying a rule. If a rule is found to be inapplicable, there are two possibilities: either there is another rule which has the same metacomponents to the left, or there is not. If there is such a rule, then that rule can be tried. If there is not, it is necessary to back-up to some previous level, and try an alternative rule at that level. It may be that ultimately the input string will be found ungrammatical; the processor will recover from unsuccessful attempts at parsing until it is back at the beginning, with no more alternatives to try. In this case it must stop, with suitable indication that the input string is bad. On the other hand, the processor may find, at some previous level, another path to follow which will lead to the successful conclusion of the parsing.

Throughout this discussion, the method in which the parsing processor keeps track of the various pieces of information that it needs has not been mentioned. Information is required both to recover from an unsuccessful attempt at applying a rule, and to construct the necessary parsing tree form of the input string for use by the unparsing processor.

There are a number of stacks which are manipulated by the processor to accomplish both of these things. The use of several of the stacks is described as follows. The stacks GOAL, INPUT, and OUTPUT are pushed down each time a new subgoal is established. They are popped up when the processing of the current subgoal is complete (whether successful or not). The stacks RULE and CONSTITUENT are pushed down each time a metacomponent is successfully recognized. They are popped up during both generation of output and recovery from an unsuccessful application of a rule. The stack LINK points to the element of OUTPUT which is the head of the subtree just successfully parsed.

The list called PARSE contains the rules generated by the parsing processor. It is essentially the Polish string form of the parsing tree. The list is generated in such a fashion that, when parsing is complete, each element of the list which corresponds to the metaresult of a rule is followed by exactly as many entries as there are metacomponents in the rule. Each entry is either a rule number, or a pointer to another section of the list which contains the appropriate rule number. The final element of the list will always be a pointer to the rule which was used to define the head of the language. Table 4 gives an example of an input string which has been parsed according to the rules stated in Table 3. Also included is the list of rule numbers with their

TABLE 4

PARSING OF A = B+C*D/E|F

| I | PARSE(I) | RULE | | | |
|---|---|---|---|---|---|
| 1 | 42 | '=' | EQ | | |
| 2 | 20 | '*' | MO | | |
| 3 | 12 | P | F | | |
| 4 | 4 | ID | P | | |
| 5 | 16 | L | ID | | |
| 6 | 25 | 'D' | L | | |
| 7 | 21 | '/' | MO | | |
| 8 | 43 | '\|' | ES | | |
| 9 | 4 | ID | P | | |
| 10 | 16 | L | ID | | |
| 11 | 27 | 'F' | L | | |
| 12 | 8 | F | ES | P | F |
| | -9 | | | | |
| | -8 | | | | |
| 15 | 12 | P | F | | |
| 16 | 4 | ID | P | | |
| 17 | 16 | L | ID | | |
| 18 | 26 | 'E' | L | | |
| 19 | 10 | T | MO | F | T |
| | -12 | | | | |
| | -7 | | | | |
| 22 | 10 | T | MO | F | T |
| | -3 | | | | |
| | -2 | | | | |
| 25 | 9 | F | T | | |
| 26 | 12 | P | F | | |
| 27 | 4 | ID | P | | |
| 28 | 16 | L | ID | | |
| 29 | 24 | 'C' | L | | |
| 30 | 7 | AO | T | UAE | |
| | -19 | | | | |
| 32 | 18 | '+' | AO | | |
| 33 | 6 | AE | UAE | AE | |
| | -30 | | | | |
| 35 | 11 | T | AE | | |
| 36 | 9 | F | T | | |
| 37 | 12 | P | F | | |
| 38 | 4 | ID | P | | |
| 39 | 16 | L | ID | | |
| 40 | 23 | 'B' | L | | |
| 41 | 1 | LP | AE | AS | |
| | -33 | | | | |
| 43 | 2 | ID | EQ | LP | |
| | -1 | | | | |
| 45 | 16 | L | ID | | |
| 46 | 22 | 'A' | L | | |
| | -41 | | | | |

associated rules. (In Appendix 6.4 is a complete set of input, including the rules of Table 3, and several source language input strings to be parsed according to those rules. Appendix 6.5 is a listing of the output of the processor corresponding to this input).

The most important features of the parsing process have now been discussed. The processor itself consists of four main sections to handle the four general situations which may arise. The sections are: (a) the initialization section, which has as its function the initialization of a defined type as a new subgoal; (b) the processing section, which attempts to build a rule chain from the current source language input string to the current subgoal, which was established by the initialization section; (c) the recovery section, which has the function of recovering from an unsuccessful attempt at applying a rule; and (d) the output section, which has the function of recording the rules used in a successful completion of a rule chain from the current characters in the input string to the current subgoal.

## 3.5 THE METASEMANTIC LANGUAGE AND PRAGMATICS

From the previous section, we have seen that parsing consists of establishing the grammatical relationships between all of the source language input string characters. The output of the parsing processor is an ordered list of all

the rules that were applied. The list elements for each rule consist of as many elements as there are metacomponents of the rule. Each element is either a rule number, or a pointer to another part of the list where the rule number will be found. This list provides a description of the structure of the input string. What is needed now is some means of attaching meaning (in terms of the target language) to this structure. This is accomplished by associating with each syntactic rule an additional semantic rule, written in a metasemantic language. The manner in which the semantic rules are used constitutes the pragmatics.

The parsing processor constructed rule chains from the bottom up, until it finally reached, as a goal, the head of the language. The output list of the parsing processor was generated in this order also, with the last element of the list being a pointer to the highest level used, that is, the rule which defines the head of the language.

The unparsing processor uses this list in the reverse order from that in which it was constructed; that is, the unparsing processor starts at the rule defining the head of the language, and progresses in a direction that is downward throught the syntax tree.

The purpose of a metasemantic language is to allow the user to state the semantics of the source language in terms of the target language. The metasemantic language developed

by Ingerman can be described fairly easily. Its use, which constitutes the pragmatics, cannot be described easily except by example. Some examples will be given in the discussion of the form of the metasemantic language.

A metasemantic rule is a construct, delimited by broken brackets '<' and '>'. The simplest construct is the null element, or <>. Another construct consists of one or more target language output characters, delimited by quotes, as <'A'>. Thus a complete metalinguistic definition of the character A as a letter is

'A' L <'A'> ,

which says that 'A' is a letter, and the base object 'A' is to be placed in the output when this rule is applied. These are considered to be "terminal" constructs, in the sense that when the processing of one of them starts, it is not necessary to process another construct in order to complete it.

A third type of construct is the "punctuation" of the output. Unless one of the punctuation symbols is used, the output character(s) will be placed in the next available position(s) in the current "line" of output. The punctuation which may be used consists of three symbols, $\alpha$, $\beta$, and $\gamma$. (For a specification of what characters are used to represent non-keypunch symbols in the implementation, consult Appendix 5, the description of the PL/I program.) The meaning of the symbols is as follows, considering the output to be produced on a standard printer:

α - prepare for characters starting in position 1 of the line (if currently past position 1, print the current line first). This is used for output of labels in symbolic machine language.

β - prepare for characters starting at position n (where n is a constant specified in the program); if past position n, print the current line first. This is used for output of symbolic operation codes.

γ - prepare for characters starting at the position which is the next largest multiple of n. This is used for output of operand fields.

The next higher type of construct is known as a clause, and provides a means for describing how a construct is associated with its environment in the parsing tree. The simplest type of clause is a direct reference to components of the metasyntactic rule with which it is associated. The reference is made by considering the elements of the metasyntactic rule to be numbered from right to left, starting with zero for the metaresult. Only defined types are numbered; base objects are omitted from the numbering scheme because they are of no semantic importance. The index numbers are then used in constructing the metasemantic rule. As an example, consider the following metalinguistic rule, which defines a statement to be a variable followed by an equal sign followed by an expression:

```
VAR  '='  EXP  ST        <(1)  β 'STORE' γ (2)>  .
(2)       (1)  (0)
```

What the metasemantic rule states is that first the chain of rules which defines EXP should be followed; then the base object 'STORE' should be put into the operation code field of the output, followed by the results of the definition of VAR in the operand field. Presumably the rule chain defining EXP will generate coding which will leave a value in a register, which will then be stored by the 'STORE' command. The rule chain defining VAR will produce an address in which to store it.

A more complicated form of the clause introduces a new kind of metasemantic element, the dummy variable. This allows a construct to define alternate meanings for a metacomponent, one of which will be selected by some lower level construct. The general form of a construct defining dummy variables is

$$<...(n,D_1...,D_2...,...)...> \quad ,$$

where n is the index of some metacomponent of the associated syntactic rule, $D_1$ is the first dummy variable followed by its definition, $D_2$ the second, followed by its definition, followed by any number of others, followed by more of the construct. What this says is essentially that if, in the following of the rule chain associated with the n-clause, the dummy variable $D_1$ is found in some lower level construct, substitute for it the definition following the $D_1$ in the

construct, and similarly for $D_2$, etc. As an example, consider the adding operators '+' and '-', the rules which define them, and a metasemantic construct which makes use of dummy variables to differentiate between them. The rule defining 'arithmetic-expression' to be a 'unary-arithmetic-expression' requires that the two operators be treated differently. In the case of the unary '+', the sign is to be ignored. In the case of the unary '-', the sign of the operand must be reversed (by a "change sign" instruction CHS, for example). The following three rules from Table 5 illustrate the use of dummy variables to make this distinction:

| | | |
|---|---|---|
| UAE | AE | <(1,A,S'CHS')> |
| '+' | AO | <A> |
| '-' | AO | <S> |

What this says is that if, in following the rule chain associated with the 1-clause, the dummy variable A is found, substitute the null element for it. If the dummy variable S is found, substitute the base objects 'CHS' for it. From this it can be seen that the possible dummy variables that may be encountered are defined in some higher level construct, and then, when one of them is encountered, the appropriate substitution is made.

Table 5 gives a listing of the rules of Table 3, with the semantic rules for the generation of appropriate symbolic machine language. Table 6 gives a description of the symbolic machine language and the machine on which it is to run. Table 7 gives the output generated by the unparsing processor for the example of Table 4.

TABLE 5

RULES OF TABLE 3 WITH SEMANTICS.

| | | | | | |
|---|---|---|---|---|---|
| 1 | LP | AE | AS | | <(1)(2)> |
| 2 | ID | EQ | LP | | <¬'STORE'%(2)> |
| 3 | ID | L | ID | | <(2)(1)> |
| 4 | ID | P | | | <¬'LOAD'%(1)> |
| 5 | UAE | AE | | | <(1,A,S¬'CHS')> |
| 6 | AE | UAE | AE | | <(2)(1,A¬'ADD',S¬'SUB')> |
| 7 | AO | T | UAE | | <(1)(2)> |
| 8 | F | ES | P | F | <(3)(1)¬'EXP'> |
| 9 | F | T | | | <(1)> |
| 10 | T | MO | F | T | <(3)(1)¬(2)> |
| 11 | T | AE | | | <(1)> |
| 12 | P | F | | | <(1)> |
| 13 | UI | D | UI | | <(2)(1)> |
| 14 | UI | P | | | <¬'LOAD'%(1)> |
| 15 | LPN | AE | RPN | P | <(2)> |
| 16 | L | ID | | | <(1)> |
| 17 | D | UI | | | <(1)> |
| 18 | '+' | AO | | | <A> |
| 19 | '-' | AO | | | <S> |
| 20 | '*' | MO | | | <'MPY'> |
| 21 | '/' | MO | | | <'DIV'> |
| 22 | 'A' | L | | | <'A'> |
| 23 | 'B' | L | | | <'B'> |
| 24 | 'C' | L | | | <'C'> |
| 25 | 'D' | L | | | <'D'> |
| 26 | 'E' | L | | | <'E'> |
| 27 | 'F' | L | | | <'F'> |
| 28 | 'W' | L | | | <'W'> |
| 29 | 'X' | L | | | <'X'> |
| 30 | 'Y' | L | | | <'Y'> |
| 31 | 'Z' | L | | | <'Z'> |
| 32 | '0' | D | | | <'0'> |
| 33 | '1' | D | | | <'1'> |
| 34 | '2' | D | | | <'2'> |
| 35 | '3' | D | | | <'3'> |
| 36 | '4' | D | | | <'4'> |
| 37 | '5' | D | | | <'5'> |
| 38 | '6' | D | | | <'6'> |
| 39 | '7' | D | | | <'7'> |
| 40 | '8' | D | | | <'8'> |
| 41 | '9' | D | | | <'9'> |
| 42 | '=' | EQ | | | <> |
| 43 | '[' | ES | | | <> |
| 44 | '(' | LPN | | | <> |
| 45 | ')' | RPN | | | <> |

TABLE 6

A Simple Stack Machine

$t_1$ denotes the top of the stack, $t_2$ the next to the top.

| Operation Code | Operation |
|---|---|
| ADD | $t_2 + t_1$ replace $t_2$, stack is popped up |
| SUB | $t_2 - t_1$ replace $t_2$, stack is popped up |
| MPY | $t_2 * t_1$ replace $t_2$, stack is popped up |
| DIV | $t_2 / t_1$ replace $t_2$, stack is popped up |
| EXP | $t_2 \uparrow t_1$ replace $t_2$, stack is popped up |
| CHS | $-t_1$ replaces $t_1$ |
| LOAD m | stack is pushed down, contents of m replace $t_1$ |
| STORE m | $t_1$ replaces contents of m, stack is popped up |

TABLE 7

Output Generated In Translation Of
A = B+C*D/E↑F

```
LOAD   B
LOAD   C
LOAD   D
MPY
LOAD   E
LOAD   F
EXP
DIV
ADD
STORE  A
```

## 3.6 THE UNPARSING PROCESSOR

The unparsing processor uses the semantic and pragmatic rules to generate the output of the translation. It must, of course, also have the output of the parsing processor, which consists of the list of rules used in the parsing (named PARSE).

The unparsing processor has a number of stacks to keep track of the current rule, the position within that rule, the level at which the processor currently is within the parsing tree (this is initially zero, and when it returns to zero, the unparsing is complete), and other information.

The basic method used is to choose a rule from the list PARSE, and then interpret the metasemantic part of the rule. It has been noted that in the list PARSE, each metaresult is followed by exactly the same number of entries as there are metacomponents of the rule. This fits in nicely with the numbering scheme used in the metasemantic language. That is, when a 1-clause is encountered in the semantic rule, the processor looks at the next sequential element of the list PARSE for the rule number to use. When a 2-clause is encountered, it looks at the second element of PARSE, and so on.

## 3.7 EXTENSIONS TO THE METALANGUAGES

Extensions to the metasyntactic language are implemented by what are called metasyntactic functions (MSF's). The general

method for implementing such MSF's requires modification of the processor to interpret them. In general an MSF is attached to a metacomponent of a rule. MSF's may be either prefixed or suffixed to the metacomponent to which they are attached. Some of the MSF's suggested by Ingerman are (a) the substitution MSF, which allows substitution of any set of base objects (or the null element) for any occurrence of a subset of the set of base objects in the input string; (b) the right context MSF, which allows re-scanning of the same input character by different defined types; and (c) the class-instance MSF, which allows recognition of instances of things which are of the same class. The only one currently implemented is the right context MSF.

An example of the use of the right context MSF is in recognizing that an 'identifier' may be terminated by a special character, as in the following rules from Appendix 2:

```
L    ID1
ID1  L    ID1
ID1  SPC  <1>  ID
```

The <1> suffixed to the metacomponent SPC is the implementation symbol for the right context MSF. Its function is to back up the input pointer after the recognition of the special character. If this were not done, the special character (an '*' for example) would be lost to the parsing processor.

Metasemantic and pragmatic functions (MPF's) allow additional meanings to be associated with syntactic rules in

more extensive ways than have been mentioned. MPF's also require modification of the processor in order to interpret them. MPF's are denoted by #n (in the current implementation n lies between 1 and 3). The MPF's which are currently implemented are (a) #1, which generates a unique label; (b) #2n, which regenerates the n-th preceding label generated by #1; and (c) #3T, which can be used to generate temporary storage labels based on the number of occurrences of the dummy variable T.

A problem encountered in translation is that of processing type declaration statements. Declaring a variable to be of a certain type is essentially a metalinguistic specification which is made by the particular source program being translated (1) (9). In order to process such a specification, it is necessary to generate rules which define the variable to be of the specified type. These rules are added to the rule set, and then another pass through the translation process is made using these rules. Included in the metasemantic language are special constructs which allow the generation of new rules.

## 3.8 COMMENTS ON INGERMAN'S METHOD

Enough of the method of translation has been discussed to be able to say a number of things about it. In general, the approach is an interesting one with the rules of syntax

and semantics being interpreted essentially in their original form by their respective processors. The metasyntactic language is sufficiently clear as to be fairly easily grasped by a new user of the translator. The notion of the acceptability table to determine the applicability of a rule is a fine one. However, there are a number of weak points in the whole process, some of which are discussed below.

From the user's point of view, the metasyntactic language is confusing at first, since the metacomponents are to the left of the metaresult, but are interpreted from left to right. It would not be difficult to change the design of the translator, or of the input routine, to allow input of rules with the metaresult on the left, and some (possibly ignored) operator which denotes the fact that what follows defines this metaresult.

As mentioned above, the idea of the ordering of the rules being done internal to the processor is a fine one. However, there are questions of the resolution of ambiguity by a certain order of rules, and the ordering algorithm may possibly change this ordering, thus changing the "meaning" of the rules. This would resolve the ambiguity in a way different from that intended by the user.

It is not clear how "reserved words" in a language should be specified. An example of a specification that works, but is clumsy, is

```
'R'   R
'R'   L
R     'E'   'A'   'L'   REAL
```

This is not a very satisfactory solution to the problem, which is a common one. Perhaps an MSF could be devised to handle this. Or perhaps some use of the substitution MSF would solve the problem.

The metasemantic language seems to be "complete" in some sense of the word. The user may produce output pretty much as he pleases. However, it is not an easy language with which to become familiar, and it is clumsy to use once one is familiar with it. Perhaps a better notation for the same ideas could be developed, which would make the language look more like something meaningful, and less like a random bunch of characters on first viewing.

The facilities for including special metasyntactic and metasemantic functions are limited. In order to include them, a specific knowledge of the processor involved is necessary. It is not clear that this could be avoided, but some thought should be given to "object time" specification of special functions, perhaps using the dynamic invocation of the PL/I compiler to implement this.

## 3.9 EXTENSIONS TO THE METHOD

From the discussion of the previous section, it is clear that there are a number of improvements that could be made to both the technique and the current PL/I implementation. Some of these are listed below.

(1) A more easily understood metalanguage which would be direct input to the rule input routine should be designed. At its simplest, this would require a few minor modifications and additions to the existing input routine. At worst, it might require a whole new approach to the problem of processing the metalanguage - a compiler for the metalanguage, perhaps.

(2) The question of what effect the automatic ordering of rules has on the definition of the structure of the source language should be investigated. If the user intends to prescribe a certain ordering for a particular resolution of ambiguity, then he should be allowed to specify that as his intent, and override the automatic ordering.

(3) The additional MSF's and MPF's that are described in Ingerman's book should certainly be implemented. They allow processing of more complicated structures such as FORTRAN DO-loops.

(4) The question of source language diagnostics from the parsing processor needs investigation. The diagnostic "BAD" is a bit too terse for most users of a translator. It would seem that at least a brief trace of the path taken in reaching the BAD point could be constructed. Perhaps, with some study, an algorithm for actually diagnosing the problem could be devised.

(5) It would be interesting to attempt to determine what languages Mr. Ingerman's scheme is sufficient for, and what additional capabilities are necessary in order to process languages of more complexity, or of more "pathological" structure. Another side of this question is what sort of pathological languages will this scheme process, and why won't it process others. As an example, FORTRAN is said to suffer "needlessly, bound in the unaccustomed corsetry" of a syntax-oriented specification (2). Can it actually be represented in the metasyntactic and metasemantic languages proposed by Ingerman? Another example might be the production of a processor for a (probably) pathological language allowing free form input for a complex statistical program.

(6) As stated before, the metasemantic language proposed by Ingerman leaves much to be desired, and much to the imagination and ingenuity of the user. The question of A more general metasemantic language, or at least a more easily usable one would be a welcome addition to the technique.

(7) An interesting project would be that of attempting to design a set of machine-independent macro-instructions for use by the unparsing processor. There has been some work done in the specification of semantics with respect to producing translators which are machine-independent. (16) A combination of this with Mr. Ingerman's metasyntactic language, and parsing and unparsing processors, might result in a very useful and general scheme which would approach the Universal Machine-Oriented Language hoped for many years ago. (14)

## 4. CONCLUSION

A general discussion of syntax-oriented translation has been presented. Some of the problems inherent both in syntactic and semantic specification and in a processor to use such specifications have been noted. A brief description of syntax-controlled methods has been given, in order to show the more general nature of syntax-oriented techniques.

Syntax-oriented methods can be used quite effectively in the study of computer languages. Once a syntax-oriented processor has been developed, it may be used to describe several source languages. The metalanguages involved can be used not only as input specifications to the processor, but also as design languages. The processor and the metalinguistic description of a source language may be used as an experimental compiler for the source language. This compiler can be easily tested, modified, and expanded, simply by changing the metalinguistic description.

However, syntax-oriented methods have certain disadvantages. A major problem is that of their inefficiency because of their generality. Another is that of providing source language diagnostics in the case of ungrammatical input.

The methods developed by Ingerman have been discussed in some detail. His translator includes a simple processor which interprets both syntactic and semantic metalanguages. This

processor works with the metalinguistic statements in essentially their original form. The processor makes use of an acceptability table to avoid tracing out unapplicable paths through the syntax tree.

It is not clear what kinds of source languages can be described using a syntax-oriented translator. It is not clear how efficient (in any sense) such a scheme might be in a production environment. What is clear is that these questions and problems cannot be investigated without more experience in using such methods.

## 5. ACKNOWLEDGEMENT

The author wishes to express her appreciation for the aid of the Washington University Computing Facilities through NIH Grants FR00215 and FR00218, and ARPA Contract SD302.

# 6. APPENDICES

APPENDIX 6.1

SPECIFICATIONS FOR META II
WRITTEN IN META II.

```
.SYNTAX PROGRAM.,
PROGRAM = '.SYNTAX' ID .OUT('CLL'*) .OUT('EXIT') '.,'
          $ ST '.END' .OUT('END').,
ST = ID .LABEL * '=' EX1 '.,' .OUT('R').,
EX1 = EX2 $ ('/' .OUT('BT'*1) EX2) .LABEL *1.,
EX2 = (EX3 .OUT('BF' *1)/OUTPUT)
          $ (EX3 .OUT('BE')/OUTPUT) .LABEL *1.,
EX3 = ID .OUT('CLL' *) /
          STRING .OUT('TST' *) /
          '.LETTER' .OUT('LET') /
          '.DIGIT' .OUT('DIG') /
          '.SPCHAR' .OUT('SPC') /
          '.QUOTE' .OUT('QU') /
          '.EMPTY' .OUT('SET') /
          '(' EX1 ')' /
          '$' .LABEL *1 EX3 .OUT('BT' *1) .OUT('SET').,
OUTPUT = ('.OUT' '(' $ OUT1 ')' /
          '.LABEL' .OUT('LB') OUT1 /
          '.DELETE' .OUT('DEL') ) .OUT('OUT').,
OUT1 = '*1' .OUT('GN1') /
          '*2' .OUT('GN2') /
          '*' .OUT('CI') /
          STRING .OUT('CL' *).,
ID = .LETTER $ (.DIGIT/.LETTER) .DELETE.,
STRING = .QUOTE (CHAR $ CHAR) .QUOTE .DELETE.,
CHAR = .LETTER/.DIGIT/.SPCHAR.,
.END
```

## APPENDIX 6.2

### Description of the INGERMAN Program

This program is a direct implementation of the general purpose syntax-oriented translator described in (1). In this description, knowledge of the technique is assumed. Only those portions of the technique which are directly concerned with the implementation will be mentioned.

Section 6.2.1 describes the current implementation with respect to what parts have not been included. Section 6.2.2 describes the character set used in the input of the metalinguistic rule set. Section 6.2.3 describes the input specifications. Section 6.2.4 describes the output of the processor.

#### 6.2.1 Details and Restrictions

Input to the processor consists of two control cards and a set of metasyntactic and metasemantic rules, followed by source language cards. The rule input routine constructs both the complete acceptability table and the extra information about the elements of R(L), as described in Chapter 3 of (1). It is planned to have the ordering algorithm included in the rule input routine, but at present it is not.

Restrictions on the form of the input are: (a) metacomponents must be four characters or less; (b) base objects must be one or two characters, delimited by quotes; (c) a syntactic rule can have at most nine metacomponents followed

by one metaresult; (d) a semantic rule can have at most 80 characters. Details of the input characters used in the implementation are given in section 6.2.2.

The complete parsing processor as described in Chapter 4 of (1) is implemented.

In Chapter 5 of (1) several additional metasyntactic functions are described. Of these, only the right context MSF is implemented.

The unparsing processor (which interprets the metasemantic language) described in Chapter 6 of (1) is fully implemented using the flow charts of Chapter 7. There are three meta-pragmatic functions (MPF's) mentioned in Chapter 6 which are implemented: ~1, ~2n, and ~3D. Implementation of the syntax generating constructs is included. However, there is as yet no provision for including the generated rules in the original rule set and making another pass over the source input.

### 6.2.2 Character Set

In (1) a notation is used which does not correspond to the PL/I character set. The following table gives Ingerman's original notation, the implementation form, and an example of use taken from Figure 6.2 of the book.

| Original | Implementation | Description/Example |
|---|---|---|
| boldface | single character in quotes | Base object as meta-component in syntactic rules. Rule [1] 'A' L |
| boldface | one or more characters in quotes | Base object(s) for output in semantic rules. Rule [40] <'MPY'> |
| lower case | one to four characters | Metacomponents in N(L) Rule [44] AO T UAE |
| <<right context>> | <1> | Right context MSF. Must be punched left adjusted in A(4) field. |
| { } | < > | Delimiters for constructs |
| [ ] | ( ) | Delimiters for clauses |
| small caps | one character | Dummy variable Rule [37] <S> |
| ! | ? | Interpretation of embedded construct. |
| α, β, γ | &, ¬, % | Punctuation |
| ϰ | # | MPF |

6.2.3 Input Specifications

There are three types of input to the processor: control cards, rule cards, and source language cards.

Control cards are punched according to the following formats.

| | Columns | Description |
|---|---|---|
| Card 1: | 1-5 | NROW: # of rows of acceptability table |
| | 6-10 | NCOL: # of columns of acceptability table |
| | 11-15 | NSTK: length of stacks |
| | 76-80 | NCDS: # of cards of source language input |

These numbers must be right adjusted within the field. They need not be exact; the estimate should be larger rather than smaller. Currently in use for a language of 116 rules are 175, 80, 80, 10.

| | Columns | Description |
|---|---|---|
| Card 2: | 1-4 | The name of the head of the language (left adjusted). |
| | 80 | 0 for no intermediate output |
| | | 1 for intermediate output |

Rule cards are punched with the syntactic rule and the semantic rule on the same card. The last rule card must have the characters ENDb punched in columns 1-4. Rule cards are read with a format of (10(A(4)),40(A(1))). The first 10 fields of 4 characters each contain the elements of the metasyntactic rule. Each element may be a base object, a name, or the right context MSF, denoted by the characters <1>. Each element must be left adjusted within the 4-character field. The meta-semantic rule is punched in columns 41-80. There may be no embedded blanks except within quotes as base object output.

If a metasemantic rule is longer than 40 characters, it must have a non-blank character in column 80. It may be continued in columns 41-80 of the next card. Columns 1-40 of the continuation card are ignored. Thus a metasemantic rule may be 80 characters long. (The current implementation will print only 74 of these characters, however.) Metasemantic rules refer to components of the corresponding metasyntactic rule. Such reference is made by numbering the elements from right to left, starting with zero for the metaresult. Base objects and the right context MSF are skipped in the numbering.

In the present implementation the rules must be sorted in the following way: (a) all rules with the same left-most metacomponent must be physically together; (b) within groups of rules with the same left-most metacomponent, the rules must be sorted so that if any two rules have the first k metacomponents identical, any rule between them must also have those first k metacomponents. In addition, if two rules have the same left-most metacomponent, the longer rule must precede the shorter one.

Source language cards are punched according to the specifications of the language under study. The processor will read them one at a time as needed, using format (80(A(1))).

An example of an input deck appears in Appendix 6.3.

6.2.4 Output

Standard output consists of a listing of (a) the rules read in; (b) the arrays P, FLAG, LMMC, COLOFA, and A (which are explained in the PL/I program listing in Appendix 6.5); (c) each source language input card as it is read; and (d) the output generated by the unparsing processor.

Optional intermediate output consists of all the above, and also (a) from the parsing processor, a printout of new subgoals as they are established, and indications of what rule is being processed, and what rule is being recovered from; (b) the list PARSE, with the rule represented by each element of PARSE; (c) from the unparsing processor, the contents of each stack each time the stacks are pushed down or popped up, the current character in the semantic rule in use, and the contents of the negative numbered rows of MATRIX (used for storage of dummy variables).

An example of standard output appears in Appendix 6.4.

## APPENDIX 6.3

SAMPLE INPUT

```
                                CARD COLUMNS
          1111111111      44444444455555555556666666      77778
1234567890123456789.....1234567890123456789012345.....67890

  175    80    80                                              10
AS                                                             0
LP   AE   AS                 <(1)(2)>
ID   EQ   LP                 <¬'STORE'%(2)>
ID   L    ID                 <(2)(1)>
ID   P                       <¬'LOAD'%(1)>
UAE  AE                      <(1,A,S¬'CHS')>
AE   UAE  AE                 <(2)(1,A¬'ADD',S¬'SUB')>
AO   T    UAE                <(1)(2)>
F    ES   P    F             <(3)(1)¬'EXP'>
F    T                       <(1)>
T    MO   F    T             <(3)(1)¬(2)>
T    AE                      <(1)>
P    F                       <(1)>
UI   D    UI                 <(2)(1)>
UI   P                       <¬'LOAD'%(1)>
LPN  AE   RPN P              <(2)>
L    ID                      <(1)>
D    UI                      <(1)>
'+'  AO                      <A>
'-'  AO                      <S>
'*'  MO                      <'MPY'>
'/'  MO                      <'DIV'>
'A'  L                       <'A'>
'B'  L                       <'B'>
'C'  L                       <'C'>
'D'  L                       <'D'>
'E'  L                       <'E'>
'F'  L                       <'F'>
'0'  D                       <'0'>
'1'  D                       <'1'>
'8'  D                       <'8'>
'9'  D                       <'9'>
'='  EQ                      <>
'|'  ES                      <>
'('  LPN                     <>
')'  RPN                     <>
END
```

```
                          CARD COLUMNS
          1111111111222222222233333333334444444444555555556
1234567890123456789012345678901234567890123456789012345678

A=B+C*D/E|F
AAA=BBB*(CCC+DDD*(EEE-FFF/(ABC|DEF)))
```

## APPENDIX 5.4

### SAMPLE OUTPUT

```
THE HEAD OF THE LANGUAGE IS:  AS

 I     SYNTACTIC                    SEMANTIC
       RULE(I)                      RULE(I)

 1     LP   AE   AS                 <(1)(2)>
 2     ID   EQ   LP                 <¬'STORE'%(2)>
 3     ID   L    ID                 <(2)(1)>
 4     ID   P                       <¬'LOAD'%(1)>
 5     UAE  AE                      <(1,A,S¬'CHS')>
 6     AE   UAE  AE                 <(2)(1,A¬'ADD',S¬'SUB')>
 7     AO   T    UAE                <(1)(2)>
 8     F    ES   P    F             <(3)(1)¬'EXP'>
 9     F    T                      <(1)>
10     T    MO   F    T             <(3)(1)¬(2)>
11     T    AE                     <(1)>
12     P    F                      <(1)>
13     UI   D    UI                 <(2)(1)>
14     UI   P                       <¬'LOAD'%(1)>
15     LPN  AE   RPN  P             <(2)>
16     L    ID                     <(1)>
17     D    UI                     <(1)>
18     '+'  AO                     <A>
19     '-'  AO                     <S>
20     '*'  MO                     <'MPY'>
21     '/'  MO                     <'DIV'>
22     'A'  L                      <'A'>
23     'B'  L                      <'B'>
24     'C'  L                      <'C'>
25     'D'  L                      <'D'>
26     'E'  L                      <'E'>
27     'F'  L                      <'F'>
28     '0'  D                      <'0'>
29     '1'  D                      <'1'>
30     '8'  D                      <'8'>
31     '9'  D                      <'9'>
32     '='  EQ                     <>
33     '|'  ES                     <>
34     '('  LPN                    <>
35     ')'  RPN                    <>
36     END
```

| I | P | FLAG | LMMC | COL. OF A | A 1111111 1234567890123456 7 |
|---|---|---|---|---|---|
| 1 | LP | 0 | 1 | 2 | 1000000000000000 |
| 2 | AE | 0 | 6 | 5 | 0000100000000000 |
| 3 | AS | 0 | 0 | 1 | 0000000000000000 |
| 4 | ID | 0 | 2 | 3 | 1111101100000000 |
| 5 | EQ | 0 | 0 | 14 | 0000000000000000 |
| 6 | L | 0 | 16 | 12 | 1111101100000000 |
| 7 | P | 0 | 12 | 4 | 0000101100000000 |
| 8 | UAE | 0 | 5 | 6 | 0000100000000000 |
| 9 | AD | 0 | 7 | 10 | 0000110000000000 |
| 10 | T | 0 | 10 | 8 | 0000100100000000 |
| 11 | F | 0 | 8 | 7 | 0000101100000000 |
| 12 | ES | 0 | 0 | 15 | 0000000000000000 |
| 13 | MD | 0 | 0 | 11 | 0000000000000000 |
| 14 | UI | 0 | 13 | 9 | 0001101110000000 |
| 15 | D | 0 | 17 | 13 | 0001101110000000 |
| 16 | LPN | 0 | 15 | 16 | 0001101100000000 |
| 17 | RPN | 0 | 0 | 17 | 0000000000000000 |
| 18 | + | 1 | 18 | 0 | 0000110001000000 |
| 19 | - | 1 | 19 | 0 | 0000110001000000 |
| 20 | * | 1 | 20 | 0 | 0000000001000000 |
| 21 | / | 1 | 21 | 0 | 0000000001000000 |
| 22 | A | 1 | 22 | 0 | 1111101100010000 |
| 23 | B | 1 | 23 | 0 | 1111101100010000 |
| 24 | C | 1 | 24 | 0 | 1111101100010000 |
| 25 | D | 1 | 25 | 0 | 1111101100010000 |
| 26 | E | 1 | 26 | 0 | 1111101100010000 |
| 27 | F | 1 | 27 | 0 | 1111101100010000 |
| 28 | 0 | 1 | 28 | 0 | 0001101110001000 |
| 29 | 1 | 1 | 29 | 0 | 0001101110001000 |
| 30 | 8 | 1 | 30 | 0 | 0001101110001000 |
| 31 | 9 | 1 | 31 | 0 | 0001101110001000 |
| 32 | = | 1 | 32 | 0 | 0000000000001000 |
| 33 | \| | 1 | 33 | 0 | 0000000000000100 |
| 34 | ( | 1 | 34 | 0 | 0001101100000010 |
| 35 | ) | 1 | 35 | 0 | 0000000000000001 |
| 36 | AE | 0 | 6 | 5 | 1000000000000000 |
| 37 | EQ | 0 | 0 | 14 | 1100000000000000 |
| 38 | L | 0 | 16 | 12 | 1111101100000000 |
| 39 | UAE | 0 | 5 | 6 | 0000100000000000 |
| 40 | T | 0 | 10 | 8 | 0000110000000000 |
| 41 | ES | 0 | 0 | 15 | 0000101100000000 |
| 42 | P | 0 | 12 | 4 | 0000101100000000 |
| 43 | MD | 0 | 0 | 11 | 0000100100000000 |
| 44 | F | 0 | 8 | 7 | 0000100100000000 |
| 45 | D | 0 | 17 | 13 | 0001101110000000 |
| 46 | AE | 0 | 6 | 5 | 0001101100000000 |
| 47 | RPN | 0 | 0 | 17 | 0001101100000000 |

SOURCE LANGUAGE INPUT
A=B+C*D/E|F

TARGET LANGUAGE OUTPUT

```
LOAD    B
LOAD    C
LOAD    D
MPY
LOAD    E
LOAD    F
EXP
DIV
ADD
STORE   A
```

SOURCE LANGUAGE INPUT
AAA=BBB*(CCC+DDD*(EEE-FFF/(ABC|DEF)))

TARGET LANGUAGE OUTPUT

```
        LOAD    BBB
        LOAD    CCC
        LOAD    DDD
        LOAD    EEE
        LOAD    FFF
        LOAD    ABC
        LOAD    DEF
        EXP
        DIV
        SUB
        MPY
        ADD
        MPY
        STORE   AAA
```

APPENDIX 6.5

LISTING OF PL/I PROGRAM

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*                                                                 */
/*          PROGRAM TO IMPLEMENT THE TRANSLATOR                    */
/*          DEVELOPED BY P. Z. INGERMAN                            */
/*                                                                 */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*                                                                 */
/*          VARIABLES GLOBAL TO ALL ROUTINES                       */
/*                                                                 */
/*    P     LIST OF ALL BASE OBJECTS AND NAMES, PLUS ALL           */
/*          NON-LMMC'S.  PARALLEL TO A.                            */
/*    LMMC  POINTER TO FIRST RULE WITH THIS ELEMENT OF             */
/*          P AS THE LMMC, OR ZERO.                                */
/*    COLOFA - POINTER TO COLUMN OF A FOR THIS ELEMENT             */
/*          OF P, OR ZERO.                                         */
/*    R     RULE MATRIX - ALL ELEMENTS ARE POINTERS TO             */
/*          P ARRAY.  BASE OBJECTS ARE < 0, NAMES > 0.             */
/*    ALABEL - POINTER TO ELEMENT OF P WHICH HEADS THIS            */
/*          COLUMN OF A.                                           */
/*    FLAG  EACH ELEMENT = 0 FOR NAME, 1 FOR BASE OBJECT           */
/*          FOR THIS ELEMENT OF P.                                 */
/*    A     ACCEPTABILITY TABLE.                                   */
/*    MROW  TOTAL NUMBER OF ELEMENTS OF P (AND ALSO LMMC,          */
/*          COLOFA, AND NUMBER OF ROWS OF A).                      */
/*    MCOL  TOTAL NUMBER OF COLUMNS OF A (AND ELEMENTS OF          */
/*          ALABEL).                                               */
/*    NUM_RULES - NUMBER OF RULES READ IN.                         */
/*    PARSE - LIST OF RULES USED IN PARSING.  INPUT TO             */
/*          UNPARSE_PART.                                          */
/*    LPARSE - POINTER TO LAST ELEMENT OF PARSE.                   */
/*    DUMPSW =0 FOR NO INTERMEDIATE PRINT-OUT, =1 FOR              */
/*          INTERMEDIATE PRINT-OUT.                                */
/*    HEAD  NAME OF HEAD OF LANGUAGE.                              */
/*    MATRIX SEMANTIC RULE MATRIX.                                 */
/*    RULES CHARACTER FORM OF SYNTACTIC RULES.                     */
/*    BADSW INITIALLY ZERO, SET TO 1 IF PARSING FAILS.             */
/*                                                                 */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
```

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*                                                                 */
/*          MAIN CONTROL ROUTINE                                   */
/*                                                                 */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
MAIN_PROGRAM: PROCEDURE OPTIONS (MAIN);
/* *       READ IN VARIOUS DIMENSIONS                             */
     GET EDIT(NROW,NCOL,NSTK,NCDS) (3(F(5)),X(60),F(5));
     CALL PROCESS;
PROCESS: PROCEDURE;
     DECLARE (P(NROW), RULES(NROW,10), HEAD) CHARACTER (4);
     DECLARE (LMMC(NROW), COLOFA(NROW), R(NROW,-3:NCOL),
          ALABEL(NCOL), PARSE(10*NSTK), MROW, MCOL,
          NUM_RULES, LPARSE) REAL FIXED BINARY;
     DECLARE (FLAG(NROW), A(NROW,NCOL), DUMPSW, BADSW)
          BIT (1);
     DECLARE FIND_IN_ ENTRY RETURNS (FIXED BINARY);
     DECLARE MATRIX(-NROW:NROW,80) CHARACTER (1);
/*                                                                 */
     CALL INPUT_PART;
CALLP: CALL PARSE_PART;
     IF BADSW THEN GO TO CALLP;
     CALL UNPARSE_PART;
     GO TO CALLP;
```

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*                                                                 */
/*          RULE INPUT ROUTINE                                     */
/*                                                                 */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
INPUT_PART: PROCEDURE;
     DECLARE TBIT BIT (1), RIN(10) CHARACTER (4);
/* *       INITIALIZATION                                         */
     P(*)='    ';
     LMMC(*)=0;
     COLOFA(*)=0;
     FLAG(*)='0'B;
     R(*,*)=0;
     ALABEL(*)=0;
     A(*,*)='0'B;
     MATRIX(*,*)=' ';
     M=0;
     N=0;
     GET EDIT (HEAD,DUMPSW) (A(4),X(75),B(1));
     PUT PAGE EDIT ('THE HEAD OF THE LANGUAGE IS ',HEAD)
          (A(28),A(4));
     PUT SKIP (2);
     LINE = 3;
     CALL HEAD1;
/* *       READ RULES AND CONSTRUCT FIRST ORDER CHAINS.           */
/*          BIG LOOP ON I FOR THE WHOLE SET OF RULES.             */
     I=1;
READ_A_RULE: GET EDIT (RIN,(MATRIX(I,J) DO J=1 TO 40))
          (10(A(4)),40(A(1)));
     IF MATRIX(I,40) ¬= ' ' THEN GET EDIT ((MATRIX(I,J)
          DO J=41 TO 80)) (X(40),40(A(1)));
     IF LINE > 55 THEN CALL HEAD1;
     PUT SKIP EDIT (I,RIN,(MATRIX(I,J) DO J = 1 TO 34))
          (F(3),X(3),10(A(4)),34(A(1)));
     LINE = LINE + 1;
     IF MATRIX(I,35) ¬= ' ' THEN DO;
          PUT SKIP EDIT ((MATRIX(I,J) DO J = 35,80))
                    (X(50),46(A(1)));
          LINE = LINE + 1;
          END;
     RULES(I,*) = RIN(*);                                        */
/*          SMALL LOOP ON J FOR EACH RULE.
     IF RIN(1) = 'END ' THEN DO;
          NUM_RULES = I-1; GO TO PART_2;
          END;
     J = 1;
J_LOOP:TBIT='0'B;
     IF SUBSTR(RIN(J),1,1)='''' THEN DO;
          IF SUBSTR(RIN(J),3,1)='''' THEN
               LEN=1; ELSE LEN = 2;
          RIN(J) = SUBSTR(RIN(J),2,LEN); TBIT='1'B;
          END;
```

```
          *** RULE INPUT ROUTINE (CONT'D) ***

/* *       LOOK FOR AND RECODE MSF'S                                    */
     IF SUBSTR(RIN(J),1,1) = '<' THEN DO;
          R(I,-3) = SUBSTR(RIN(J),2,1);
          R(I,-2) = J-1;
          DO K = J+1 TO 10; RIN(K-1) = RIN(K); END;
          GO TO J_LOOP;
          END;
     MROW = M;
     K = FIND_IN_P(RIN(J),TBIT);
/* *       PUT NEW METACOMPONENT IN P ARRAY IF NOT THERE.               */
     IF K = 0 THEN DO;
          M=M+1;
          P(M)=RIN(J);
          FLAG(M)=TBIT;
          K=M;
          END;
IN_P: IF J=1 & LMMC(K)=0 THEN LMMC(K)=I;
     IF TBIT='1'B THEN K=-K;
     R(I,J)=K;
     IF J<NCOL & RIN(J+1)¬='    ' THEN DO;
          J=J+1; GO TO J_LOOP;
          END;
/* *       THE END OF THIS RULE - FINISH IT UP                          * */
     R(I,0)=J;
     K = ABS(K);
/* *       SET UP THE COLUMN OF A FOR THIS META-RESULT                  * */
     IF COLOFA(K)=0 THEN DO;
          N = N+1;
          ALABEL(N)=K;
          COLOFA(K)=N;
          END;
     IP=COLOFA(K);
/* *       SET FIRST ORDER RULE CHAIN FROM LMMC TO M-R                  * */
     IQ = ABS(R(I,1));
     A(IQ,IP)='1'B;
     I=I+1;
     GO TO READ_A_RULE;
```

```
          *** RULE INPUT ROUTINE (CONT'D) ***

/* *       ENTER NON-LEFT M-C'S INTO P AS NEW COMPONENTS  */
PART_2: DO I = 1 TO NUM_RULES;
     IF R(I,0) > 2 THEN DO J = 2 TO R(I,0)-1;
          IF R(I,J)>0 THEN DO;
               M=M+1;
               K=R(I,J);
               P(M)=P(K);
               FLAG(M)=FLAG(K);
               LMMC(M)=LMMC(K);
               COLOFA(M) = COLOFA(K);
               R(I,J)=M;
               A(M,COLOFA(R(I,R(I,0)))) = '1'B;
               END;
          END;
     END;
/* *       FINISH UP THE EXTRA ROWS OF THE A MATRIX      * */
     DO J = 1 TO N;
     IP = ALABEL(J);
     DO I = 1 TO M;
     IF I¬=IP & A(I,J)='1'B THEN A(I,*) = A(I,*)|A(IP,*);
          END;
     END;
     MROW = M;
     MCOL = N;
/* *       INTERMEDIATE PRINT OUT                        * */
     LINE = 56;
     NN = (N/2)+3;
     DO I=1 TO M;
     CALL HEAD2;
     PUT SKIP EDIT(I,P(I),FLAG(I),LMMC(I),COLOFA(I),
          (A(I,J) DO J=1 TO N))
          (F(5),X(5),A(4),B(1),2(F(5)),
          X(5),100(B(1)));
     END;
     RETURN;
```

*** RULE INPUT ROUTINE (CONT'D) ***

```
/* *       ROUTINE TO PRINT FIRST HEADINGS                          */

HEAD1: PROCEDURE;
     IF LINE > 55 THEN DO; PUT PAGE; LINE = 0; END;
     PUT SKIP EDIT ('I','SYNTACTIC','SEMANTIC')
          (X(1),A(1),X(4),A(9),X(31),A(8));
     PUT SKIP EDIT ('RULE(I)','RULE(I)')
          (X(6),A(7),X(33),A(7));
     PUT SKIP (2);
     LINE = LINE + 3;
     RETURN;
     END HEAD1;

/* *       ROUTINE TO PRINT SECOND HEADINGS                         */

HEAD2: PROCEDURE;
     LINE = LINE + 1;
     IF LINE <= 55 THEN RETURN;
     PUT PAGE;
     PUT SKIP EDIT ('COL.','A')(X(23),A(4),X(NN),A(1));
     PUT SKIP EDIT ('I','P','FLAG','LMMC','OP',
          (K/10 DO K = 10 TO N))
          (X(4),A(1),X(5),A(1),X(2),A(4),X(1),A(4),X(2),
          A(2),X(13),70(F(1)));
     PUT SKIP EDIT ('A',(MOD(K,10) DO K=1 TO N))
          (X(24),A(1),X(5),80(F(1)));
     PUT SKIP;
     LINE = 5;
     RETURN;
     END;

     END INPUT_PART;

/* *       ROUTINE TO FIND ELEMENTS OF P                            * */

FIND_IN_P:  PROCEDURE (ITEM,TESTBIT) FIXED BINARY;
     DECLARE ITEM CHARACTER(*), TESTBIT BIT(1);
     DO IP = 1 TO MROW;
     IF P(IP) = ITEM & FLAG(IP) = TESTBIT THEN RETURN (IP);
     END;
     RETURN(0);
     END FIND_IN_P;
```

```
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
/*                                                               */
/*          PARSING PROCESSOR                                    */
/*                                                               */
/*                                                               */
/* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * */
PARSE_PART: PROCEDURE;
     DECLARE (GOAL(NSTK), INPUT(NSTK),OUTPUT(NSTK),
          NULL(NSTK), EXIT(4*NSTK), CONST(4*NSTK),
          REF(4*NSTK),LINK(2*NSTK),RULE(4*NSTK))
          FIXED BINARY REAL;
     DECLARE (TEMP, TEMP2) REAL FIXED BINARY;
     DECLARE S(80*NCDS) CHARACTER(1);
     DECLARE CTEMP CHARACTER(4), IS REAL FIXED BINARY;
     DECLARE NULLRC BIT(1);
     BADSW='0'B;
     PUT PAGE EDIT('SOURCE LANGUAGE INPUT') (A(21));
     PUT SKIP (2);
     CTEMP = '';
     KNULL = FIND_IN_P(CTEMP,'1'B);
/*        FIND HEAD OF LANGUAGE IN P - IHEAD IS POINTER  */
     IHEAD = FIND_IN_P(HEAD,'0'B);
/*        READ AN INPUT CARD                             */
     IS = 1;
     CALL GET_S;
/*        FIND S(1) IN P - I IS POINTER                  */
     I = FIND_IN_P(S(1),'1'B);
     J=COLOFA(IHEAD);
/*        SET FLAG TO 1 FOR RULE CHAIN FROM '' TO HEAD   */
     NULLRC = '0'B;
     IF KNULL ¬= 0 THEN NULLRC = '1'B;
/*        IS THERE A RULE CHAIN FROM S(1) TO HEAD?   */
     IF ¬A(I,J) THEN GO TO TEST_NULL_1;
/*        YES - INITIALIZE   */
INIT:  TEMP = J;
     IG,INP,IO,IN,IE,IR ,IC,IRF,IL = 1;
     RULE(IR),CONST(IC) = 0;
     INPUT(INP),OUTPUT(IO) = 1;
     EXIT(IE) = 4;       /*   FLAG FOR END OF STATEMENT   */
     REF(IRF),LINK(IL),NULL(IN) = 0;
     GO TO SET_GOAL;
```

*** PARSING PROCESSOR (CONT'D) ***

```
/*          INITIALIZATION SECTION                                  */

NUMBER_1:
/*          PRESERVE GOAL, INPUT, OUTPUT, NULL                      */
     IG=IG+1;  GOAL(IG) = GOAL(IG-1);
     INP = INP+1;  INPUT(INP) = INPUT(INP-1);
     IO = IO+1;  OUTPUT(IO) = OUTPUT(IO-1);
     IN = IN+1;  NULL(IN) = NULL(IN-1);
SET_GOAL:  GOAL(IG) = TEMP;
     IF DUMPSW THEN PUT SKIP EDIT ('GOAL IS ',
          P(ALABEL(GOAL(IG))), '  LEVEL IS ', IG)
          (A(8),A(4),A(11),F(5));
/*          IS THE INPUT CHARACTER THE LMMC OF SOME RULE?  */
     I = FIND_IN_P(S(INPUT(INP)),'1'B);
     IF I=0 THEN GO TO TEST_NULL_2;
     J = LMMC(I);
     IF J = 0 THEN GO TO TEST_NULL_2;
     RULE(IR) = J;
/*          IS THERE A RULE CHAIN FROM INPUT TO GOAL?    */
     IF ¬A(I,GOAL(IG)) THEN GO TO TEST_NULL_2;
/*          YES,INDEEDY / GO LOOK AT THIS (SUB-)GOAL    */
     NULL(IN) = 0;
     CALL UP_INPUT;
     GO TO NUMBER_4;

/*          IS THERE A RULE CHAIN FROM NULL TO HEAD?        */
TEST_NULL_1:   IF NULLRC THEN GO TO INIT;
     PUT SKIP EDIT('NUL1') (A(4));
     GO TO END2;
TEST_NULL_2:      IF ¬NULLRC THEN GO TO NUMBER_3;
     IF NULL(IN)=0 THEN GO TO NUMBER_3;
#2A: NULL(IN) = 1;
     K = LMMC(KNULL);
     GO TO NUMBER_4;
```

*** PARSING PROCESSOR (CONT'D) ***

```
/*          PROCESSING SECTION                                          */

NUMBER_4: CONST(IC) = 2;
/*          IS R(R,C) IN N(L)?          */
#4A:  IF DUMPSW THEN PUT SKIP EDIT ('PROCESS', RULE(IR),
          CONST(IC), RULES(RULE(IR),*))
          (X(10),A(7),2(F(5)),X(5),10(A(4)));
      IF R(RULE(IR),CONST(IC))>0 THEN GO TO #4B;
/*          NO - IT'S IN B(L)                                           */
      K = ABS(R(RULE(IR),CONST(IC)));
/*          IS R(R,C) THE INPUT CHARACTER?          */
      IF S(INPUT(INP)) ¬= P(K) THEN GO TO NUMBER_3;
/*          YES          */
      CALL UP_INPUT;
NUMBER_6:  CONST(IC) = CONST(IC) + 1;
      GO TO #4A;
/*          R(R,C) I S IN N(L) - IS IT A META-RESULT?        */
#4B:  IF  CONST(IC) < R(RULE(IR),0) THEN GO TO #4C;
/*          YES - IS THERE A RULE CHAIN FROM HERE TO GOAL?  */
      K = R(RULE(IR),CONST(IC));
      IF A(K,GOAL(IG)) THEN GO TO #4D;
/*          NO - ARE WE AT THE GOAL?                                    */
NUMBER_5:  IF R(RULE(IR),CONST(IC)) = ALABEL(GOAL(IG))
          THEN GO TO NUMBER_7;
          ELSE GO TO NUMBER_3;
/*          META-COMPONENT - IS THERE A RULE                            */
/*          CHAIN FROM HERE TO GOAL?                                    */
#4C:  K = R(RULE(IR),CONST(IC));
      IF ¬A(K,GOAL(IG)) THEN GO TO NUMBER_3;
/*          YES - GO A NEW SUB-GOAL, SO                                 */
/*          PRESERVE STUFF AND GO BACK.                                 */
      IE = IE+1;  EXIT(IE) = 5;
      IR = IR+1;  RULE(IR) = RULE(IR-1);
      IC = IC+1;  CONST(IC) = CONST(IC - 1);
      IRF = IRF+1;  REF(IRF) = LINK(IL);
      TEMP = COLOFA(K);
      GO TO NUMBER_1;                                                 */
/*          RULE CHAIN FROM HERE TO GOAL
#4D:  IE = IE+1;  EXIT(IE) = 6;
      IR = IR+1;  RULE(IR) = LMMC(R(RULE(IR-1),CONST(IC)));
      IC = IC+1;  CONST(IC) = CONST(IC-1);
      IRF = IRF+1;  REF(IRF) = LINK(IL);
      GO TO NUMBER_4;
```

*** PARSING PROCESSOR (CONT'D) ***

```
/*          RECOVERY SECTION                                          */

/*          ARE ALL CONSTITUENTS TO THE LEFT OF THIS                  */
/*          ONE EQUAL TO THE CONSTITUENTS                             */
/*          OF THE NEXT RULE?                                         */
NUMBER_3:  IF DUMPSW THEN PUT SKIP EDIT ('RECOVER FROM',
           RULE(IR), CONST(IC)) (X(10),A(12),2(F(5)));
      DO J=1 TO CONST(IC)-1;
      JRP1 = ABS(R(RULE(IR)+1,J)); JR = ABS(R(RULE(IR),J));
      IF JRP1 ¬= JR THEN IF P(JRP1) ¬= P(JR) THEN GO TO #3A;
      END;
      RULE(IR) = RULE(IR) + 1;
      IF R(RULE(IR),CONST(IC)) = R(RULE(IR)-1,CONST(IC))
           THEN GO TO NUMBER_3;
           ELSE GO TO #4A;
#3A:  IF NULL(IN) ¬=0 THEN GO TO #3B;
           ELSE IF ¬NULLRC THEN GO TO #3B;
      GO TO #2A;
#3B:  IF REF(IRF) ¬= LINK(IL) THEN DO;
           IL = IL-1;
           GO TO #3B;
           END;
      EXIT(IE) = EXIT(IE) - 3;
NUMBER_9:  TEMP2 = EXIT(IE);
      IF DUMPSW THEN PUT SKIP EDIT ('TEMP2=',TEMP2)
           (X(10),A(6),F(5));
      IE = IE-1;  IR = IR-1;  IC = IC-1;  IRF = IRF-1;
      IF TEMP2 = 1 THEN GO TO BAD;
      IF TEMP2 = 2 THEN DO;
           IO = IO-1; INP = INP-1; IG = IG-1;
           IN = IN-1; GO TO NUMBER_3;
           END;
      IF TEMP2 = 3 THEN GO TO NUMBER_5;
      IF TEMP2 = 4 THEN DO;
           PARSE(OUTPUT(IO)) = TEMP; GO TO END_IT_ALL;
           END;
      IG = IG-1; IN = IN-1;
     /*  LOSE OUTPUT  */
      IO = IO-1;  OUTPUT(IO) = OUTPUT(IO+1);
/*          LOSE INPUT, UNLESS THERE'S A RIGHT                        */
/*          CONTEXT MSF, IN WHICH CASE JUST RESTORE INPUT             */
      INP = INP-1;
      IF ¬((R(RULE(IR),-3)=1) & (R(RULE(IR),-2)=CONST(IC)))
           THEN INPUT(INP) = INPUT(INP+1);
      IL = IL+1; LINK(IL) = TEMP;
      GO TO NUMBER_6;
```

*** PARSING PROCESSOR (CONT'D) ***

```
/*          OUTPUT SECTION                                          */

NUMBER_7: TEMP = -OUTPUT(IO);
#7A:   PARSE(OUTPUT(IO)) = RULE(IR);
#7B:   OUTPUT(IO) = OUTPUT(IO) + 1;
       IF DUMPSW THEN PUT SKIP EDIT ('OUTPUT', RULE(IR),
            OUTPUT(IO)-1) (X(10),A(6),2(F(5)));
       IF REF(IRF) ¬= LINK(IL) THEN DO;
            PARSE(OUTPUT(IO)) = LINK(IL);
            IL = IL - 1; GO TO #7B;
            END;
       IF EXIT(IE) ¬= 6 THEN GO TO NUMBER_9;
       IE = IE-1; IR = IR-1; IC = IC-1; IRF = IRF-1;
       GO TO #7A;
```

*** PARSING PROCESSOR (CONT'D) ***

```
/*          RETURN SECTION                                          */

BAD:  PUT SKIP EDIT ('BAD')  (A(3));
      BADSW = '1'B;
      GO TO END2;

END_IT_ALL:
END2:  LPARSE = OUTPUT(10);
      IF LPARSE = 1 THEN RETURN;
      IF ¬DUMPSW THEN RETURN;
      PUT PAGE;
      DO I = 1 TO LPARSE;
      K = PARSE(I);
      IF K<0 THEN DO; PUT SKIP EDIT (PARSE(I))(F(10)); END;
            ELSE DO; PUT SKIP EDIT (I,PARSE(I),
            (RULES(K,J) DO J = 1 TO 10),
            (MATRIX(K,J) DO J = 1 TO 60))
            (2(F(5)),X(5),10(A(4)),60(A(1)));
            END;
      END;
      RETURN;


/*          PARSING UTILITY SUBROUTINE                              */

UP_INPUT: PROCEDURE;
      INPUT(INP) = INPUT(INP) + 1;
      IF INPUT(INP) <= IS+79 THEN RETURN;
      IS = IS+80;
GET_S: ENTRY;
      GET EDIT ((S(I) DO I = IS TO IS + 79)) (80(A(1)));
      PUT SKIP EDIT ((S(I) DO I=IS TO IS+79)) (80(A(1)));
      RETURN;
      END UP_INPUT;

      END PARSE_PART;
```

```
          *** UNPARSING PROCESSOR (CONT'D) ***

NUMBER_4A:  INSIDE(PTR) = INSIDE(PTR) + 1;  T = 0;
NUMBER_4: CALL SUBC;
     K = CT(PTR) + CHAR;
     CALL SUBC;
     IF CHAR ¬= ',' THEN GO TO #0;
     OUTPUT = '0'B;
     GO TO #5A;
#5:  IF CHAR = ',' THEN DO;
          IF P ¬= 0 THEN GO TO NUMBER_3;
          CALL SUBF;
#5A:      CALL SUBC;
          NEG(PTR) = NEG(PTR) - 1;
          EXTR1(NEG(PTR),1) = 2;
          EXTR1(NEG(PTR),2) = LEV(PTR);
          EXTR1(NEG(PTR),3) = CT(PTR);
          MATRIX(NEG(PTR),1) = CHAR;
          CALL DUMPM;
          GO TO #3;
          END;
     IF CHAR = ')' THEN
          IF P = 0 THEN DO; CALL SUBF; GO TO #0; END;
          ELSE DO; P = P-1; GO TO NUMBER_3; END;
     IF CHAR = '(' THEN P = P+1;
     GO TO NUMBER_3;
#6:  T3 = NEG(PTR);
     IF SUBG = '0'B THEN DO;
NUMBER_3: CALL SUBB;
          GO TO #4;
          END;
     CALL PRESERVE;
     LEV(PTR) = EXTR1(T3,2);
     ROW(PTR) = T3;
     COL(PTR) = 1;
     CT(PTR) = EXTR1(T3,3);
     GO TO #2;
```

```
          *** UNPARSING PROCESSOR (CONT'D) ***

NUMBER_1: INSIDE(PTR) = 1;
NO_1A:    CALL SUBB;
NO_1B: IF CHAR = '<' THEN DO;
          INSIDE(PTR) = INSIDE(PTR)+1; GO TO NO_1A;
          END;
     IF CHAR = '>' THEN DO;
          DO I = COL(PTR)+1 TO 80;
          IF MATRIX(ROW(PTR),I) ¬= ' ' THEN GO TO NO_1B1;
          END;
          GO TO #4;
NO_1B1:   INSIDE(PTR) = INSIDE(PTR) - 1;
          IF INSIDE(PTR) = 0 THEN GO TO NUMBER_3;
               ELSE GO TO NO_1A;
          END;
NO_1C: IF CHAR = '?' THEN DO;
          CALL SUBC;
          T = T+1;
          IF T¬= INSIDE(PTR) THEN GO TO NO_1C;
          IF CHAR ¬= '(' THEN DO;
               CALL SUBH; GO TO NUMBER_2;
               END;
          IF OUTPUT THEN GO TO NUMBER_4A;
          END;
     IF T¬= 0 THEN DO;
          CT3=CHAR;
          CHAR = '?';
NO_1D:    T = T-1;
          CALL SUBE;
          IF T¬=0 THEN GO TO NO_1D;
          CHAR = CT3;
          END;
     CALL SUBE;
NUMBER_2: CALL SUBA;
     GO TO NO_1B;
```

*** UNPARSING PROCESSOR (CONT'D) ***

```
/*          UNPARSING PROCESSOR SUBROUTINES                          */
SUBA: PROCEDURE;
SA1:  CALL SUBC;
      IF CHAR = '''' THEN DO;
           CALL SUBE;
SA15:      CALL SUBC;
           IF CHAR = '''' THEN DO;
                CALL SUBE; GO TO SA1;
                END;
           CALL SUBE;
           GO TO SA15;
           END;
      IF CHAR = '&' THEN GO TO SUBB1;
      IF CHAR='%' THEN GO TO SUBB1;
      IF CHAR='¬' THEN GO TO SUBB1;
      RETURN;
SUBB: ENTRY;
SUBB1: CALL SUBE;
      GO TO SA1;
      END SUBA;

SUBC: PROCEDURE;
      COL(PTR) = COL(PTR) + 1;
      IF COL(PTR) > 80 THEN DO;
           PUT SKIP EDIT ('OOPS',ROW(PTR)) (A(4),F(5));
           GO TO END_UNPARSE;
           END;
      CHAR = MATRIX(ROW(PTR),COL(PTR));
      IF DUMPSW = '0'B THEN RETURN;
      PUT SKIP EDIT ('CHAR=', CHAR) (A(5),A(1));
      RETURN;
      END SUBC;
```

*** UNPARSING PROCESSOR (CONT'D) ***

```
SUBD:  PROCEDURE;
SD1: T2 = EXTR1(NEG(PTR),1);
     MATRIX(NEG(PTR),T2) = CTEMP;
     EXTR1(NEG(PTR),1) = T2+1;
     CALL DUMPM;
     RETURN;
SUBE: ENTRY;
    IF ISW ¬= 0 THEN DO;  INP = INP+1;
          IF ISW = 1 THEN NAME1(INAME) =
               SUBSTR(NAME1(INAME),1,INP-1)||CHAR;
          IF ISW = 2 THEN NAME2(INAME) =
               SUBSTR(NAME2(INAME),1,INP-1)||CHAR;
          IF ISW = 3 THEN TNAME =
               SUBSTR(TNAME,1,INP-1)||CHAR;
          RETURN;
          END;
     IF ¬OUTPUT THEN DO; CTEMP=CHAR; GO TO SD1; END;
     IF CHAR = '''' THEN RETURN;
     CALL OUTRTN(CHAR);
     RETURN;
     END SUBD;

SUBF: PROCEDURE;
     CTEMP = '>';
     CALL SUBD;
     RETURN;
     END SUBF;

SUBG: PROCEDURE BIT(1);
SG1: IF T3 = 0 THEN RETURN('0'B);
     IF CHAR = MATRIX(T3,1) & LEV(PTR) > EXTR1(T3,2)
          THEN RETURN ('1'B);
     T3 = T3 + 1;
     GO TO SG1;
     END SUBG;
```

```
          *** UNPARSING PROCESSOR (CONT'D) ***

SUBH: PROCEDURE;
      DECLARE BLAH CHARACTER(9);
      DECLARE TLAB CHARACTER(4) INITIAL('TLAB');
      CALL SUBC;
      IF CHAR = '1' THEN DO;
           LABEL(PTR) = LABEL(PTR)+1;
           BLAH = LABEL(PTR);
SUBH1:     CHAR = '''';
           CALL SUBE;
                DO I = 1 TO 4;
                CHAR = SUBSTR(TLAB,I,1);
                CALL SUBE;
                END;
           GO TO SUBHEND;
           END;
      IF CHAR = '2' THEN DO;
           CALL SUBC; JJJ = LABEL(PTR)-CHAR;
           BLAH = JJJ; GO TO SUBH1;
           END;
      IF CHAR = '3' THEN DO;
           T3 = NEG(PTR);
           CALL SUBC; IDUM = SUBG;
           CHAR = '''''; CALL SUBE;
           BLAH = ABS(T3);
SUBHEND:   DO I = 1 TO 4;
           CHAR = SUBSTR(BLAH,I+5,1);
           IF CHAR ¬= ' ' THEN CALL SUBE;
           END;
           CHAR = ''''; CALL SUBE;
           RETURN;
           END;
      IF CHAR = '4' THEN DO; CALL SUBC;
           IF CHAR = '1' THEN DO;
                ISW = 1; INAME = INAME+1;
                INP = 0; RETURN;
                END;
           IF CHAR = '2' THEN DO;
                ISW = 2; INP = 0; RETURN;
                END;
           IF CHAR = '3' THEN DO;
                CHAR = '>'; CALL SUBE;
                ISW = 0; RETURN;
                END;
           END;
```

```
      *** UNPARSING PROCESSOR (CONT'D) ***

IF CHAR = '5' THEN DO; CALL SUBC;
      IF CHAR = '1' THEN DO;
           ISW = 3; TNAME = ' ';
           INP = 0; RETURN;
           END;
      IF CHAR = '2' THEN DO; ISW = 0;
           DO I = 1 TO TNAME;
           IF TNAME = NAME1(I) THEN DO J = 1 TO 20;
             CHAR = SUBSTR(NAME2(I),J,1);
             IF CHAR = '>' THEN RETURN;
             CALL SUBE;
             END;
           RETURN;
           END;
      RETURN;
      END;
      END;
RETURN;
END SUBH;
```

*** UNPARSING PROCESSOR (CONT'D) ***

```
/*          UNPARSING UTILITY SUBROUTINES                          */

PRESERVE:   PROCEDURE;
     PTR = PTR+ 1;
     INSIDE(PTR) = INSIDE(PTR-1);
     ROW(PTR) = ROW(PTR-1);
     COL(PTR) = COL(PTR-1);
     LEV(PTR) = LEV(PTR-1);
     CT(PTR) = CT(PTR-1);
     NEG(PTR) = NEG(PTR-1);
     LABEL(PTR) = LABEL(PTR - 1);
     IF DUMPSW = '0'B THEN RETURN;
     PUT SKIP EDIT(PTR,K,INSIDE(PTR),ROW(PTR),COL(PTR),
          LEV(PTR),CT(PTR),NEG(PTR),LABEL(PTR))
          (9(F(10)));
     RETURN;
     END PRESERVE;

RESTORE:   PROCEDURE;
     PTR = PTR-1;
     IF DUMPSW = '0'B THEN RETURN;
     PUT SKIP EDIT(PTR,K,INSIDE(PTR),ROW(PTR),COL(PTR),
          LEV(PTR),CT(PTR),NEG(PTR),LABEL(PTR))
          (9(F(10)));
     RETURN;
     END RESTORE;
```

```
          *** UNPARSING PROCESSOR (CONT'D) ***

OUTRTN:  PROCEDURE(CHAR);
     DECLARE CHAR CHARACTER(1), IPTR INITIAL (0);
  /*   ¬ IS BETA,  % IS GAMMA,  & IS ALPHA                    */
     IF CHAR = '¬' THEN
          IF ILOC <= ITAB THEN DO;
               ILOC = ITAB; RETURN;
               END;
          ELSE DO; IPTR = ITAB; GO TO OUT2;
          END;
     IF CHAR = '%' THEN DO;
          IPTR = ILOC/ITAB; ILOC = (1+IPTR)*ITAB;
          IPTR = 4*ITAB; GO TO OUT1;
          END;
     IF CHAR = '&' THEN GO TO OUT2;
     ILOC = ILOC+1;
     LINE(ILOC) = CHAR;
OUT1:  IF ILOC < 120 THEN RETURN;
OUT2:  PUT SKIP EDIT (LINE) (120(A(1)));
     LINE = ' ';
     ILOC = IPTR;
     RETURN;
     END OUTRTN;

DUMPM: PROCEDURE;
     IF DUMPSW = '0'B THEN RETURN;
     PUT SKIP EDIT(NEG(PTR), (EXTR1(NEG(PTR),J)
          DO J = 1 TO 3), (MATRIX(NEG(PTR),J)
          DO J = 1 TO EXTR1(NEG(PTR),1)))
          (X(40),4(F(5)),40(A(1)));
     RETURN;
     END;

END_UNPARSE: IF ILOC > 0 THEN CALL OUTRTN('&');
     RETURN;
     END UNPARSE_PART;
     END PROCESS;
     END MAIN_PROGRAM;
```

## 7. BIBLIOGRAPHY


1. Ingerman, P. Z., "A Syntax-Oriented Translator", Academic Press, New York and London, 1966.

2. Floyd, R. W., "The Syntax of Programming Languages - A Survey", IEEE Transactions on Electronic Computers, August 1966.

3. Naur, P., "Revised Report on the Algorithmic Language Algol 60", Communications of the ACM, Vol. 6, No. 1, (1-17), January 1966.

4. Schorre, D. V., "META II - A Syntax-Oriented Compiler Writing Language", ACM Proceedings of the 19th National Conference, August 1964.

5. Schneider, R. W. and Johnson, G. D., "META III - A Syntax-Directed Compiler-Writing Compiler to Generate Efficient Code", ACM Proceedings of the 19th National Conference, 1964.

6. Ledley, R. S. and Wilson, J. B., "Automatic-Programming-Language Translation Through Syntactical Analysis", Communications of the ACM, Vol. 5, No. 3, (145-155), March 1962.

7. Irons, E. T., "A Syntax Directed Compiler for Algol 60", Communications of the ACM, Vol. 4, No. 1, (51-55), January 1961.

8. Floyd, R. W., "A Descriptive Language for Symbol Manipulation", Journal of the ACM, Vol. 8, (579-584), 1961.

9. Cheatham, T. E., Jr., and Sattley, K., "Syntax-Directed Compiling", Proceedings of the SJCC, Spartan Books, Baltimore, Maryland, Vol. 25, (31-57), 1964.

10. Davis, Ruth M., "Programming Language Processors", Advances in Computers, Vol. 7, (117-180), Academic Press, New York and London, 1966.

11. Graham, R., "Bounded Context Translation", Proceedings of the SJCC, Spartan Books, Baltimore, Maryland, Vol. 25, (17-29), 1964.

12. Floyd, R. W., "Syntactic Analysis and Operator Precedence", Journal of the ACM, Vol. 10, No. 7, (316-333), July 1963.

13. Randall, B. and Russell, L. J., "Algol 60 Implementation", A.P.I.C. Studies in Data Processing No. 5, Academic Press, New York, 1964.

14. "The Problem of Programming Communication with Changing Machines - Part 1", Communications of the ACM, Vol. 1, No. 8, (12-18), August 1958.

15. Irons, E. T., "An Error-Correcting Parse Algorithm", Communications of the ACM, Vol. 6, No. 11, (669-673), November 1963.

16. Feldman, J. A., "A Formal Semantics for Computer Languages and Its Application in a Compiler-Compiler", Communications of the ACM, Vol. 9, No. 1, (3-9), January 1966.

## 8. VITA

Biographical items on the author of the thesis, Mrs. Laetitia H. Snow:

1) Born January 30, 1936.

2) Received the degree of Bachelor of Science in Mathematics, from the University of North Carolina, in June, 1958.

3) Programmer, Defense Systems Department, General Electric, Syracuse, New York, June 1958 to June 1959.

4) Programmer, Electric Boat Division, General Dynamics Corporation, Groton, Connecticut, August 1959 to May 1961.

5) Systems programmer, Applied Mathematics Department, Monsanto Chemical Company, St. Louis, Missouri, December 1961 to June 1963.

6) Systems programmer, Computing Facilities, Washington University, St. Louis, Missouri, June 1963 to present.

7) Attended Washington University in the Department of Applied Mathematics and Computer Science, Sever Institute of Technology, from September 1962 to present.

8) Member of A.C.M.

June, 1967